

"Quite simply, this book is a must-have for any serious MFC developer."

—Dean McCrory, MFC Development Lead

MFC Internals

*Inside the Microsoft® Foundation
Class Architecture*

GEORGE SHEPHERD
SCOT WINGO



MFC Internals

**Inside the Microsoft® Foundation
Class Architecture**

George Shepherd and Scot Wingo

Foreword by Dean D. McCrory



ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters or all capital letters.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data

Shepherd, George (George R.), 1962--

MFC internals : inside the Microsoft Foundation class architecture

/ George Shepherd and Scot Wingo.

p. cm.

Includes index.

ISBN 0-201-40721-3

1. Microsoft Windows (Computer file) 2. Microsoft foundation
class library 3. C++ (Computer program language) I. Wingo,

Scot. II Title.

QA76.76.W56S4967 1996

005.26'2—dc20

95-52102

CIP

Copyright © 1996 by George Shepherd and Scot Wingo

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.
Published simultaneously in Canada.

Sponsoring Editor: Claire Horne

Project Manager: John Fuller

Production Coordinator: Ellen Savett

Cover design: Jean Seal

Set in 11-point Times by Rob Mauhar, CIP of Coronado

Text printed on recycled and acid-free paper.

ISBN 0201407213

1314151617 MA 06 05 04

12th Printing August 2003

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact: Pearson Education Corporate Sales Division, One Lake Street, Upper Saddle River, NJ 07458, (800) 382-3419, corpsales@pearsontechgroup.com

To Kris, for putting up with the long hours, CT years,
loud music, and me—and, most importantly, for being a great
friend and wife.

—S.W.

To my mother, Elizabeth Shepherd, and to the memory of
my father, George R. Shepherd.

—G.S.

Contents

Foreword xvii

Acknowledgments xix

Introduction xxi

Why MFC Internals? xxi

MFC Internals to the Rescue! xxii

Who Will Benefit from This Book xxiii

How to Use This Book xxiii

A Word of Caution xxiv

Contacting the Authors xxv

Chapter 1 A Conceptual Overview of MFC 1

Some Background on Object-Oriented Programming 1

OOP Terminology 2

Abstraction 2

Encapsulation 2

Inheritance 2

Polymorphism 3

Modularity 3

Objects in General 3

Objects and C++ 4

Why Use OOP? 6

Application Frameworks and MFC 7

A Little History 7

Design Goals for MFC 8

The First Release 10

But That Wasn't Enough . . .	10
Where Are We Now?	11
A Grand Tour of MFC	12
General-Purpose Classes	12
Windows API Classes	16
Application Framework Classes	21
High-Level Abstractions	22
Operating System Extensions	24
Conclusion	28
Chapter 2 Basic Windows Support	29
MFC versus C/SDK	30
All That (Boilerplate) Code	31
Comparing the Code	32
The WinMain() Function	36
Initializing a Particular Instance of the Application	37
The Message Loop	37
Message Handling	38
Basic MFC Application Components	39
CWinApp: The Application Object	39
CWnd: The Base Window Class	42
Turning Window Handles into Window Objects	44
Attaching and Detaching Window Handles	46
Find WinMain() Now	46
MFC State Information	48
Back to WinMain()	51
Initializing the Framework: AfxWinInit()	52
Some Other Hidden Cool Stuff	54
Registering Window Classes	54
MFC's Windows Hooks	57
MFC's Message Pump: CWinApp::Run()	58
MFC's GDI Support	58
Device Contexts	59
Graphic Objects	61
Conclusion	63

Chapter 3 Message Handling in MFC	65
CCmdTarget and Message Maps	65
Window Messages	66
Message Handling Using C and the SDK	67
Windows and C++	69
MFC Message-Mapping Internals	70
The CCmdTarget Class	70
Message Map Data Structures	70
Message Map Macros	71
How MFC Uses Message Maps	75
How MFC Windows Become Wired to a WndProc	76
Handling Messages	77
Handling WM_COMMAND	81
Handling Regular Window Messages	87
Other Kinds of Messages	89
Hooking into the Message Loop: PreTranslateMessage()	91
Conclusion	92
Chapter 4 The MFC Utility Classes	93
Simple Value Types	93
Class CString: A char * on Steroids	94
Other Simple Value Types	106
MFC Collection Classes	107
MFC Collection Shapes	108
MFC Array Collections	108
Lists	114
MFC Map Collections	122
The CFile Family: MFC Access to Files	129
Using CFile	130
CFile Internals	131
CStdioFile	133
CMemFile	136
There Is Another . . .	141
CFile Summary	142
CException: Providing Better Error Handling	143
CException Internals	145
Conclusion	148

Chapter 5 All Roads Lead to CObject	151
Isn't That Expensive?	151
CObject Features	151
An Introduction to Macros	153
Run-Time Class Information	153
RTCI: How Does It Work?	155
Dynamic Creation	159
What's Left to Learn about CRuntimeClass?	161
Persistence in MFC	162
Adding Serialization to Your Classes	162
Serialization: How It Works	163
CObject and Serialization	169
Tracing a Write and a Read of a CObject Derived Object	170
Serialization Performance	173
A CRuntimeClass Status Update	174
CObject Diagnostic Support	174
Diagnostic Output	174
Run-Time Checking	175
Memory Diagnostics	177
Inside CObject Diagnostic Support	180
Diagnostic Output	180
Advanced Memory Diagnostics	187
Back to CMemoryState	192
Cool AFX Helper Functions	195
Putting It All Together	196
Putting It into Practice	197
Is It Worth It?	198
Conclusion	199
Chapter 6 MFC Dialog and Control Classes	201
CDialog: Modal and Modeless MFC Dialogs	201
Using CDIALOG	201
CDIALOG Internals	204
CDIALOG Control Initialization	212
DDX/DDV: CDIALOG Exchange and Validation	215
MFC Common Dialogs	225
Using the MFC Common Dialog Classes	226
Common Dialogs Internals	227

MFC Common Dialog Wrapup	235
OLE Dialogs	236
Using the OLE Dialogs	237
MFC OLE Dialog Class Internals	237
OLE Dialog Summary	239
Property Sheets (a.k.a. Tabbed Dialogs)	239
Using MFC Tabbed Dialogs	240
Property Sheet and Page Internals	241
Property Sheet Recap	249
MFC Control Classes	250
The "Old-Fashioned" Windows Control Classes	250
The "New-Fangled" MFC Windows Common Control Classes	256
Conclusion	257
Chapter 7 MFC's Document/View Architecture	259
Why You Want Document/View	259
Other Reasons	260
The Old Way	260
The Architecture	261
Documents and Views	261
Document/View Components	262
CWinApp's Role	266
Inside the Document/View Architecture	267
The CWinApp/CDocTemplate Interface: CDocManager	267
CDocTemplate: CDocument, CView, and CFrameWnd Manager	273
CFrameWnd Internals	279
CDocument Internals	282
CView Internals	288
Document/View Internals Recap	289
Document/View Interdependencies	289
Creation Information	290
Putting It All Together	290
Conclusion	293
Chapter 8 Advanced Document/View Internals	295
Mirror, Mirror, on the Wall . . .	295
Inside CMirrorFile	296

CView Printing	299
CView Printing Overview	300
CView Printing Internals	302
Inside CView Print Preview Support	307
CView::OnFilePrintPreview()	308
CPrintPreviewState	309
Inside DoPrintPreview()	309
CPreviewView: An Undocumented Print Preview CView Derivative	311
Inside CPreviewView::SetPrintView()	314
Print Preview Wrapup	316
CView Derivatives: CScrollView	317
How CScrollView Works	317
CScrollView Recap	323
CFormView: Forms in a View	324
CFormView Recap	327
Another CView Derivative: CCtrlView	327
CCtrlView: How It Works	327
CTreeView: An Example Control View/CCtrlView Derivative	329
CCtrlView Wrapup	330
Conclusion	331
Chapter 9 MFC's Enhanced User-Interface Classes	333
CSplitterWnd: MFC Splitter Windows	333
The Anatomy of a Splitter Window	334
Refresher: How to Use CSplitterWnd	335
Inside CSplitterWnd	336
CSplitterWnd Recap	363
The MFC CControlBar Architecture	365
Step Right Up to the CControlBar	365
CControlBar Persistence	382
CControlBar Layout Management	390
CControlBar Recap	391
CMiniFrameWnd	392
MFC MRU File List Implementation	394
How MFC Implements MRU File Lists	394
Conclusion	396

Chapter 10 MFC DLLs and Threads	399
Understanding States	399
MFC States Explained	400
How the MFC States Are Related	405
MFC DLLs	407
DLL Resource Issues	408
Extension DLL Initialization and Cleanup	414
AFXDLL and Macros	414
MFC Threads	417
MFC Worker Threads	417
MFC User-Interface Threads	424
A Tale of Threads, Handles, and Objects	429
Conclusion	429
For More Information	430
Up Next	431
Chapter 11 How MFC Implements COM	433
MFC and OLE	434
The Component Object Model	434
What Is a COM Class?	435
Do You Speak I-Speak?	437
Globally Unique Identifiers	439
Exploring the Great IUnknown	439
Object Lifetime Management	440
Interface Negotiation	442
A Peek at the Client Side: Call/Use/Release	443
COM Object Servers	444
In-Proc Servers (DLLs)	445
Out-of-Proc Servers (EXEs)	445
Class Factories	446
Exposing the Class Factory in an In-Proc Server	449
Exposing the Class Factory in an Out-of-Proc Server	449
Unloading In-Proc Servers	450
Unloading Out-of-Proc Servers	451
Class Registration in the Registry	452
Creating Instances of COM Classes	453
COM Classes with Multiple Interfaces	454
Implementing CoMath Using Multiple Inheritance	455

Implementing CoMath Using Nested Classes	458
A Class Factory for CoMath	462
MFC COM Classes	464
Using MFC to Create CoMath	467
IUnknown and CCmdTarget	467
COM Aggregation	468
The Internal IUnknown Functions	473
The External IUnknown Functions	474
Multiple Interfaces through Nested Classes	474
The MFC COM and Interface Map Macros	475
Declaring the Nested Classes	475
Building the Interface Map	477
The CoMath Class Using MFC	479
MFC COM Classes and Inheritance	482
InternalQueryInterface() Revisited	484
Finishing the Server	485
Implementing the Interfaces	485
MFC Support for Class Factories	487
COleObjectFactory	487
Developer Tip: Registering Other Information	491
The Heart of COleObjectFactory: OnCreateObject()	493
COleObjectFactory and IClassFactory::LockServer()	493
Creating Class Factories within Your App	494
COleObjectFactory and Interface Maps	495
Exporting the Class Factory from a DLL	497
Conclusion	498
Chapter 12 Uniform Data Transfer and MFC	501
Some History	501
The Old Way	502
Limitations of the Windows Clipboard and DDE	503
OLE and Uniform Data Transfer	504
Important Structures	504
The FORMATETC Structure	504
The STGMEDIUM Structure	506
The IDataObject Interface	507
IDataObject::GetData()	508

IDataObject::GetDataHere()	508
IDataObject::QueryGetData()	508
IDataObject::GetCanonicalFormatEtc()	508
IDataObject::SetData()	508
IDataObject::EnumFormatEtc()	509
IDataObject::DAdvise()	509
IDataObject::DUnadvise()	509
IDataObject::EnumDAdvise()	509
The OLE Clipboard	509
MFC's IDataObject Classes	511
Transferring Data via the Clipboard	511
Delayed Rendering	515
MFC's IDataObject Classes in Detail	516
COleDataSource	517
COleDataObject	523
OLE Drag-and-Drop	526
IDropSource	526
IDropTarget	526
Implementing Drag-and-Drop Data Transfer Using MFC	528
Originating a Drag-and-Drop Transfer	528
Implementing a Drop Target	529
Inside MFC's Drag-and-Drop Classes	531
How MFC Drag-and-Drop Works	534
Conclusion	538
Chapter 13 OLE Documents the MFC Way	539
OLE Documents 101	540
Linking and Embedding	540
Structured Storage, Compound Files, and Persistent Objects	542
In-Place Activation and Visual Editing	547
OLE Document Containers	548
OLE Document Servers	549
The OLE Document Protocol	550
MFC's Support for OLE Documents	550
The Base Classes: CDocItem and COleDocument	550
OLE Document Containers the MFC Way	552
COleLinkingDoc	554

OLE Document Servers the MFC Way	557
COleServerDoc	557
COleServerItem	561
The Container/Server Dance (Embedding)	561
The Container: Creating a New File	561
Adding Container Items	562
Deactivating the Item	575
Inside COleClientItem::Close()	575
Saving the Container's Document	577
Loading OLE Documents	578
Conclusion	580

Chapter 14 MFC and Automation	581
The History of Automation	581
What Automation Can Do for You	582
Writing an MFC Automation Application	584
But How Does It All Work?	584
COM Interfaces versus Automation	585
COM Interfaces Reviewed	585
IDispatch: The Key to Automation	587
Implementing IDispatch by Hand	591
Another Way: Using Type Information	597
Type Information	597
Object Description Language	597
Implementing IDispatch Using Type Information	599
Recap: Automation in the Raw	601
MFC and Automation	602
Extensions to CCmdTarget	602
Dispatch Maps	604
CCmdTarget and GetIDsOfNames()	611
CCmdTarget and Invoke()	613
MFC and Type Information	614
Conclusion: The Consequences of "The MFC Way"	617

Chapter 15 OLE Controls	619
VBXs and Their Shortcomings	620
OLE Controls	620

Writing a Control	621
Using OLE Controls in a Project	622
OLE Controls in a Dialog Box	622
Using an OLE Control in a View	623
So How Does It All Work?	624
MFC's OLE Control Classes	624
MFC and OLE Control Containers	628
COleControlContainer	628
COleControlSite	628
COccManager	629
A Day in the Life of an OLE Control	629
Control Creation	630
Inside the Control	632
Finishing Creating the Control	633
OLE Connections	634
OLE Connection Interfaces	635
Establishing a Connection	636
Establishing a Connection (Continued)	639
OLE Control Events	639
Firing Events	641
MFC OLE Control Events	641
How MFC Handles Events	642
Developer Tip: Adding an Event Sink to a View	643
Adding a Function to Handle the Event	644
Setting Up the Event Sink Map	645
Inside OLE Control Property Pages	646
Property Pages in a Nutshell	647
Programming the Property Page	648
Inside the Properties Verb	649
Inside COlePropertyPage	651
Accessing the Properties	652
Inside GetPropCheck()	653
Property Page Wrapup	653
Conclusion	654
Appendix A A Field Guide to the MFC Source Code	655
MFC Coding Techniques	656
Class Declaration Subsections	656

Variable Naming—Think Hungarian!	657
Symbol-Naming Conventions	658
The Proof Is in the Pudding!	658
Common MFC #defines	659
Granularity and Swap Tuning?!	660
Tools for Exploring MFC	661
The Visual C++ Browser	661
Visual C++ Find in Files	662
A Visual C++ Syntax-Coloring Tip	662
Commercial Products	662
A Guide to the MFC Source Code	662
MFC Directory Structure	662
MFC Header Files	664
MFC Inline Files	672
MFC Resources	673
MFC Source Files	674
Happy Trails . . .	687
<i>Appendix B The MFC Internals Floppy</i>	689
<i>Index</i>	691

Foreword

by **Dean D. McCrory**

Since MFC 1.0 was released in 1990, the Microsoft Foundation Classes (currently at version 4.0) have grown into a full-featured, robust, highly rated, highly used, and very complex application framework. For those of us who have been along for the ride, there is a lot of history and experience with previous versions of MFC that help mitigate this complexity. For those programmers just starting with MFC today, there is an incredible learning curve. Reaching MFC nirvana requires persistence, writing code using MFC, and plenty of reading.

Of course, Visual C++ (and other C++ products that license MFC) comes with both reference and conceptual information to describe the MFC library. There are also many third-party books that describe how to create Windows applications with MFC. All of these sources, however, concentrate on how you use MFC: what to call, what to override, and how to enable certain features. Although the necessity to understand the MFC implementation beyond the documentation has been debated in a few interesting conversations (flame wars?) on several popular online forums, it is my opinion that to become a true expert in MFC you need to understand the MFC implementation. The final answer to any question can always be found in the source code itself. This is one of the primary reasons that the source code for MFC is provided.

Studying the source code was, in fact, how I first learned MFC; I did so in part from necessity and part because of my nature. I became involved in MFC during an internal “alpha” release that was made available midway in the development of Microsoft C/C++ 7.0. At that time there was no documentation for MFC, and even CodeView didn’t work for C++ code yet. The lack of documentation didn’t bother me, because I really wanted to understand how *everything* worked. I kept asking questions like, “How is it that MFC calls my C++ member functions for Windows messages?” and “How is MFC’s exception handling implemented?” (At the time, Microsoft’s C++ did not support exception handling in the compiler.) So, I started

browsing through the code attempting to understand it. But should everyone using MFC have to do this? Does everyone have that much time? Probably not—some people have a life!

Even if you do have enough time to plow through the piles of MFC source code, this book can help guide you through what's there. Scot and George have done an impressive job analyzing the MFC code and providing explanations for many critical MFC implementation details in this book. In fact, I've been approached a number of times to write this kind of book, but I've been too busy cranking out new versions of MFC to spend the amount of time required to create something like this book. Fortunately, I did have time to review this entire book and provide a few insights.

This book is definitely not a rehash of documents that already exist. It is not a “how-to” book but a “how-does-it-work” book. Many sections start like this: “The following declaration has been stripped of all the documented member functions, so we can concentrate on the undocumented implementation details.” These implementation details let you understand some of the issues or limitations that may be inherent in a given class, even if they are not apparent in the class's public interface. Knowledge of these details and how they affect your application can affect how you use the class in your own code. These are important details when it comes down to creating “world class” applications. Besides, some of the details are just plain interesting or may show C++ techniques you'll want to use in your own code (or techniques you'll want to avoid). The authors do not expose all the interesting stuff, however. Some investigation is left up to the reader. And after reading much of the analysis in this book, you'll likely obtain skills that allow you to better investigate topics not covered here. Building these skills is very important—many of the questions asked on the popular on-line services (even questions asked on Microsoft internal MFC and Visual C++ aliases) can be answered by a quick detour through the MFC code.

Quite simply, this book is a *must-have* for any serious MFC developer. In fact, this book will be required reading for both new and old members of the MFC development, quality assurance, and user-education teams inside Microsoft.

Acknowledgments

Scot Wingo: First, thanks to you the reader for your interest in this kind of book: it was a blast to write. Thanks to Claire Horne for signing up someone who had no experience writing a book or with MFC (just kidding!). Thanks to Dean McCrory for all of his insights, funny AFX-team stories, and, of course, raw technical feedback.

Special thanks to my dog Mack for providing moral support while every page was written, and for the frequent Frisbee breaks. (The truth is that he actually writes all of this stuff. I just watch).

Finally, thanks to my fellow Stingrays, for being great partners and pals; my parents, for birthing me; the folks at 1203 Belle Mead, for all the great chili-fests; the Coca-Cola Company, for inventing and distributing Mello Yello—the MFC Internals caffeinated beverage of choice; Superchunk, Small, StereoLab, and Juliana Hatfield, for writing great MFC Internals tunes; Dad, Todd McFarlane, John Walker, Frank Miller, and Jim Lee, for being great entrepreneurial role models; and last, but certainly not least, George, for letting me on board and for being a great person to collaborate with on this project.

George Shepherd: Whenever I read books about software and programming, the authors inevitably say something like “Thanks to my spouse for putting up with my absence while I wrote this book.” I thought they were kidding, or just being nice. Now I know differently. Completing this book was one of the hardest things I’ve ever done. Indeed, thanks are due to my wife Sandy for the patience she had whenever I would disappear up to my office to toil away on this project. In addition, Sandy was a valuable reviewer, helping me straighten out poorly written or confusing text. Now that I’ve got some clock cycles back, I can begin to repay her for all the time I was absent and for all the help she gave me.

Thanks to my son Teddy for understanding why I couldn’t go to the zoo sometimes.

Thanks to Don Box, for explaining OLE in a way that makes sense; to Mary Kirtland, for helping me get this project rolling; to Andrew Schulman, for the initial

feedback and for recommending that Addison-Wesley go forward with this book; to Jeff Duntemann, for publishing my first two articles and for helping me get my start in writing; to J.D. Hildebrand and the folks at Oakley Publishing, for their help and encouragement; to Claire Horne at Addison-Wesley, for her patience as we hammered out chapters (not always as fast as she would have liked); to Dean McCrory, for editing the technical aspects of the book; to my brother Patrick, for being a sounding board for ideas; and to my parents, for pushing me to take the extra effort to always do the best I can. Finally, I'd like to thank Scot Wingo for being such a great and enthusiastic writing partner.

Introduction

Another book on the Microsoft Foundation Classes? What gives?

As you browse in the C++ and Windows sections of your favorite bookstore, you'll find a lot of books about Microsoft Visual C++ and the Microsoft Foundation Classes (MFC). Most of them are MFC cookbooks that focus on the Wizard code-generation tools and cover the basics of using MFC. This book is different—you won't find another one like it. Unlike the others, this one is about MFC's internals. It tells the story of how MFC is put together and what goes on behind the scenes.

Why MFC Internals?

Why would you need a book on MFC internals? To answer that question, let's look at a typical MFC programming scenario.

Using AppWizard, you click some buttons, make some choices, and within ten minutes you have a working Windows application that supports the Multiple Document Interface (MDI) and Object Linking and Embedding (OLE) and has a floating toolbar, a status bar, printing, print preview, and an about box. Early in the MFC experience, you have in fact become Super Programmer—faster than a speeding C programmer and able to leap OLE in a single bound!

Then you decide that you want to modify your skeleton application to do something like handle multiple document types: for example, adding a GIF/BMP image viewer. Suddenly you hit a brick wall. Where does MFC keep the list of documents? How does it go about adding their type to the File Open dialog? Your first impulse is to check the MFC documentation, but you find that it does not go into such details. Luckily, Microsoft ships the MFC source code with the libraries, so all you have to do is look around in there and figure it out, right?

Wrong. Soon you realize that MFC is a pretty hefty body of code, with over 120,000 lines in the source files alone, not to mention the header files and H-extension files. So you spend the next week or so adding a feature that seems as if it should take minutes—especially after your AppWizard experience. You've lost that fleeting Super Programmer feeling and now spend your days (and nights) pulling your hair in search of MFC source code understanding and knowledge of how to make MFC bend according to *your* will.

There are several other occasions when you need to know more about what's going on inside MFC. Here are some examples that we have encountered while programming with MFC:

- When debugging your application, you frequently land right in the middle of MFC source code, surrounded by unfamiliar classes and undocumented internal structures. Wouldn't it be nice to know what these undocumented MFC structures are doing and how they affect the problem you are tracing?
- In MFC, many of the classes are meant for derivation only. In other words, you don't create instances of them directly, but you create a derivative and instantiate your derivative. In the process, you have to override virtual functions from the base class to get the desired behavior in your derivative. The problem is that the MFC documentation is not always clear about when you should completely replace the function and when you should first call the function and then add your own code. For example, if you create a CDialog derivative and want to overload OnInitDialog() to do some initialization, should you call the base CDialog::OnInitDialog() first, or can you just completely replace the function without worrying about the implications? Obviously, to make this decision you should understand what CDialog::OnInitDialog() is doing—only then will it be obvious whether or not to call the base class function.
- MFC's OLE support is fairly complete, but if you ever want to extend it or write your own COM objects, you will leave the safety of the documented MFC OLE classes and have to figure out the MFC OLE code on your own.

MFC Internals to the Rescue!

Though the Microsoft Foundation Classes hide the detail and organize the functionality of Windows code very well, they do not eliminate the need to understand what is going on under the hood. If you want to create sizzling, commercial-grade apps that do more than just throw around dialog boxes, you have to know how MFC

works. There is a lot of stuff going on inside MFC. Much of it is not obvious, and some of it is not even documented. That's where this book comes in.

We've combed through the MFC source code searching for useful information that can help you get over such MFC hurdles. We've uncovered a plethora of interesting tidbits, such as undocumented classes, utility functions, and data members; interesting C++ techniques; useful coding techniques; and of course tons of details about the way the various MFC classes work and how they all fit together. We'll put all of the key facts and critical findings that our research has revealed into this book so that you can quickly solve your MFC programming problems (and get back to being Super Programmer again!).

Who Will Benefit from This Book

This book is written for serious Windows developers who have decided to use the Microsoft Foundation Classes, version 2 or higher. (Well, we don't really care if you're serious, as long as you're willing to learn.)

We assume two things about you as the reader:

1. You understand C++. MFC is a C++ class library, so you need to grasp the way C++ works. You need to be able to read the code, follow the syntax, and understand the basic ideas underlying object-oriented programming, such as—yep, you guessed it—encapsulation, inheritance, and polymorphism. MFC relies heavily on these concepts in its implementation. Along the way, if we find any C++ technique used by MFC that is particularly advanced or just plain interesting, we'll be sure to point it out, so that you can start using the same techniques.
2. You have a basic grasp of Windows from a development point of view. You need to understand concepts such as window handles, messages, device contexts, rectangles, and so on.

How to Use This Book

MFC Internals focuses on the 32-bit MFC version 4.0 for Windows 95 and Windows NT. If you are using an older version of MFC, don't worry: most of the internal concepts have been pretty constant over the years. Whenever this is not the case, we will be sure to warn you with a version note.

Throughout the book, we shine a bright floodlight on important concepts that are missing from the standard documentation. Along the way we point out any undocumented classes, features, member functions, and so on, and show you how to use them in your own applications. You will find some examples of how to use MFC internals on the accompanying diskette.

The first half of this book (Chapters 1–8) focuses on the core Windows graphical user interface classes and their supporting classes. In the second half of the book (Chapters 9–14), we cover subjects such as OLE that are extensions to the basic Windows support. Finally, in Chapter 15 we present advanced MFC internals that do not fit under a single topic.

Our goal in designing this book was flexibility. You should be able to read the book from cover to cover or use it as a reference. In either case, if you've never looked at the MFC source code, Appendix A, "A Field Guide to MFC Source Code," might be a good place to start; it reviews the MFC source code organization and covers some of the Microsoft coding guidelines.

If you are a beginning MFC programmer, you will probably want to start with Chapter 1, an overview of MFC, and then read Chapters 2 and 5 before skipping around the rest of the book. Chapter 2 shows you how MFC encapsulates Windows, and Chapter 5 shows you how the granddaddy of most MFC classes, CObject, works and what it adds to the equation.

If you are an intermediate MFC programmer, you can probably review the first five chapters and then skip to the topics you find the most interesting. For example, if you need some information on how the MFC document/view architecture is implemented, you could jump right to Chapter 7.

Finally, if you are an advanced MFC programmer, you might want to focus on the chapters that cover the internals you have not explored yet.

A Word of Caution

There's usually a good reason that Microsoft hasn't documented many of the areas we expose: they expect them to change. In fact, throughout the writing of the book, we've had to revise various chapters because MFC changes so quickly (four releases a year!). If you decide to use any of the internals that we highlight in your applications, be sure to comment or #ifdef them, so that if a new version of MFC breaks your usage, you will know what's going on. For example, in previous versions of MFC, the lists of documents and views were kept right in the document template. In version 4.0, Microsoft moved these lists to a new class, CDocManager. If you used the previous location of the lists, you would have to rewrite that code to use the new class.

Contacting the Authors

We've done our best to present only information that is timely and factual. If you find any problems in the book, have any suggestions, or just want to discuss a topic, you can visit our World Wide Web page at http://www.stingsoft.com/mfc_internals and send us feedback. This site will also contain any revisions to the book and pointers to other MFC sites of interest. If you do not have Web access, you can drop us an e-mail at mfc_internals@stingsoft.com; we will try to get back to you as soon as possible.

And now, venture forth.

A Conceptual Overview of MFC

If you understand what the designers of MFC were trying to accomplish, you'll see the entire MFC framework more clearly. And grasping the framework allows you, the programmer, to take greater advantage of MFC's capabilities. This chapter gets you started by presenting a high-level conceptual overview of MFC and by introducing you to some MFC history. We begin with a review of the basic tenets of object-oriented programming (OOP) and an outline of the design goals for MFC. (If you're familiar with OOP's principles, then skip ahead to the section "Application Frameworks and MFC.") Then we present the big picture: MFC's components and how these basic parts fit together.

Some Background on Object-Oriented Programming

These days, it seems as if the ads for every software package claims that their programs use objects. Some really are object oriented, and some are not. But what does it mean for something to be object oriented? MFC is truly object oriented. To understand MFC thoroughly, you need to understand the fundamentals of OOP, including such concepts as *abstraction*, *encapsulation*, *inheritance*, *polymorphism*, and *modularity*. This section provides general background on OOP and describes these object-oriented concepts.

Structured design, analysis, and programming was de rigueur from the mid-seventies through the late eighties. It's what everyone learned in Computer Science 101. It also helped produce countless systems, many of which are still in use today. But as software developers continue to take on larger and more ambitious projects, structured techniques are beginning to show cracks. That's not to say that they are now

useless, but the larger and more complicated systems being built today require some new techniques and ideas to help manage their complexities. OOP models problems in the real world in ways not possible using structured techniques. As a result, many developers are adopting object-oriented analysis, design, and programming to help meet the challenges of managing such huge and complex systems.

OOP Terminology

The terms generally used to describe an object-oriented system are *abstraction*, *encapsulation*, *inheritance*, *polymorphism*, and *modularity*. A programming language must support all these features before it deserves to be called object oriented. Following is a more detailed description of each of these concepts.

Abstraction

An abstraction defines the essential characteristics of an object that distinguish it from other objects, relative to the viewer's perspective. An abstraction looks at an object from the outside while ignoring details about how the object is implemented. Abstractions define contracts, or protocols, for interaction between classes of objects. Selecting the proper abstractions, or classes, is the major focus of object-oriented design.

Encapsulation

Encapsulation is the ability to combine code and data into one integral unit. Abstraction and encapsulation go hand in hand. Without encapsulation, you couldn't define the kind of abstractions that model real-world objects. In C++, the units of encapsulation are *classes* and *structs*. Ideally, you should be able to use an object without ever looking at its implementation. Revealing the implementation increases the complexity because you must deal with it as a client of the class.

However, this is not an ideal world. As Grady Booch, author of *Object-Oriented Analysis and Design*, puts it: "In practice, there are times when one must study the implementation of a class to really understand its meaning, especially if the external documentation is lacking." (That's why this book on MFC exists!)

Inheritance

Inheritance is the ability to derive new classes from existing ones. Inheritance lets you build class hierarchies that help you leverage your existing code. In C++, you

can easily derive one class from another, thereby inheriting all the functionality of the base class. Once you have all that good functionality in hand, you change or add to it according to your needs.

Polymorphism

Polymorphism literally means “having many shapes.” In OOP, this means that a class can share a specific function name up and down the class hierarchy, though each specific class in the hierarchy exhibits the behavior in its own particular style. For example, imagine developing some shape-drawing classes using OOP. Each shape would have its own drawing method. Even though each object’s drawing method looks the same, you definitely want each shape to exhibit its own drawing behavior. You want the circle object to draw circles, and you want the square object to draw squares. The ability of each object to respond in its own way to the drawing method is an example of polymorphism. In C++, polymorphism is accomplished via *virtual member functions*. Virtual member functions are bound to their classes at run time instead of compile time (like regular C functions). Polymorphism can occur in C++ because the decision about which member function to call is made at run time.

Modularity

Though classes form the components of a system, they alone are not sufficient to compose a whole system. To define a system’s architecture, you need to group classes into modules. Modules are essential for managing complexity (especially for larger systems), so determining how to group your classes into modules is almost as difficult as selecting the classes in the first place. Modularity is essentially a kind of encapsulation that occurs when you group related classes together to provide some higher-level behavior. A good module is functionally cohesive and provides the smallest possible interface that satisfies the needs of its clients.

Objects in General

An object is something that has attributes and exhibits behavior. Consider the computer you use every day. It’s an object: it has attributes and exhibits behavior. Even though the inner workings of your computer are very complex, that doesn’t keep you from using it. There is a *layer of abstraction* between you and the computer. The implementation details of your computer are safely hidden away inside the metal,

plastic, and silicon that make up the machine, but your interaction with it is very clearly defined. That is, the machine is *encapsulated*. You send it messages (for example, turn it on, press a key, move the mouse) and it produces behavior in response (it boots up, a letter appears on the screen, the mouse cursor moves). What's more, your computer *inherits* attributes and behaviors from other computers that came before it, thereby forming a *hierarchy*. For example, the Pentium-based computer you have on your desk today is derived from a long line of 80x86-based machines. Along the way, each new machine provides basically the same features as the ones before it, but with some additional features. The progression of PCs forms a hierarchy, with each new machine inheriting properties from the previous generation.

Most computers today perform many of the same basic fundamentals, even though the underlying implementation is different. This is known as *polymorphism*. For example, the IBM-PC is not the only brand of personal computer available. Think about your friend with a Macintosh. His Macintosh shares most of the same attributes and behaviors of the IBM machine, but each machine implements the behaviors in its own way. For instance, the Macintosh and the IBM-compatible PC both respond to mouse movements, but the underlying implementation is somewhat different. This is an example of polymorphism in action.

Finally, your computer is made up of other objects. The CPU, the disk drives, the display adapter, and so on are objects themselves. You can think of your computer as a module comprising all these separate objects. This is an example of modularity.

Objects and C++

In software, an object is a data entity that knows what to do with itself. It's a way of gathering together related data and functions that operate on that data in one package. As in the real world, software objects are entities that you can treat as abstractions. That is, they are things that have attributes and behaviors and a clearly defined interface.

Here's an example: Suppose you want to create a small program in C to keep track of people's ages. One way you could approach the task is to define a structure:

```
typedef struct _PERSON{
    char szName[25];
    int nAge;
} PERSON;
```

and then define some functions to initialize and show a person's data:

```
void SetData(PERSON *pPerson, char *pszName, int nAge);
void ShowData(PERSON *pPerson);
```

Notice what's happening here. On one hand, there are the person records, and on the other hand there are functions that operate on the data. They're separate entities. C++ changes all this by combining data and functions into a single entity—a C++ *class*. A class is very much like a C struct except it has the functions that operate on the data melded together with the data. The Person structure can be defined as a class like this:

```
class Person {
    char m_szName[25];
    int m_nAge;
public:
    void SetData(char *pszName, int nAge);
    virtual void ShowData();
};
```

Notice that the functions are now part of the class. SetData() and ShowData() are called the class's *member functions*.

Note: C++ structs can also have member functions. In C++, the difference between a class and a struct is the default visibility of the member variables and functions. Members of C++ structures are public by default, whereas C++ class members are private by default.

This ability to combine the data structure with functions is called encapsulation, and it's an essential attribute of any object-oriented programming language. Once you define the Person class and implement its member functions, initializing and displaying the data becomes merely a matter of creating an instance of the class and calling the appropriate member functions:

```
main()
{
    Person person;
    person.SetData("Sam", 218);
    person.ShowData();
}
```

But that's not all you can do with a class. C++ also lets you derive new classes from existing classes, thereby introducing a class that already has the same data and exhibits the same behavior as the ancestor class. This property, called inheritance, is another essential attribute of an object-oriented programming language. Using the Person database example, suppose you want to create a new kind of person called an employee. An employee shares all the same attributes and behaviors

as a person, plus a new attribute called “occupation.” In C++, you can derive an Employee class from a Person class by doing the following:

```
class Employee : public Person {
    char m_szOccupation[25];
public:
    void SetData(char *pszName, int nAge, char *pszOccupation);
    virtual void ShowData();
};
```

At this point, the Employee class inherits all the attributes and behaviors of the Person class, in addition to having its own new data member, m_szOccupation. This is an example of inheritance, a vital part of any object-oriented language.

Notice that there is a new way to show the employee data. This is required to account for the “occupation” member variable. Because the Person class doesn’t know anything about occupation, the Employee class needs to redefine the ShowData() member function to include showing the employee’s occupation. This is an example of polymorphism, which means that objects can share a function name up and down a class hierarchy, with each class in the hierarchy implementing the function in a way appropriate to itself.

Of course, the Employee class needs to live somewhere. This is where modularity comes in. In C++, modularity is accomplished using header files and source code files. Class definitions and other information needed to understand a C++ class is published in a header file. The actual implementation code is included in one or more source files. In this example, the Employee class definition might go in a file called EMPLOYEE.H while the actual implementation might go in a file called EMPLOYEE.CPP.

Although this example offers a simplified view of abstraction, encapsulation, inheritance, polymorphism, and modularity, these basic concepts are the heart of object orientation.

Why Use OOP?

Using object-oriented techniques to develop software helps to construct systems that closely model the real world. The real world is not made up of data acted upon capriciously by outside sources. The real world is made up of all kinds of autonomous entities communicating with each other, such as a company consisting of employees working together. That’s what objects do inside a program. Each object knows how to handle its job very well, and it collaborates with other objects (that also know how to do their jobs very well) to accomplish a common goal.

Using C++ as a real object-oriented language (that is, not as just a “better C than C”) and applying object-oriented techniques encourages the reuse not only of code, but also of designs. This speeds the development process and reduces long-term maintenance costs. Systems built using the object model are generally more resilient to change as well, because they are built upon objects that model real-world objects in the problem space. Finally, using the object model reduces risk by encouraging an evolutionary software development process, in which independent components are gradually integrated to build increasingly complex systems. One particularly useful use of OOP is to create reusable application frameworks.

A framework is a well-engineered collection of classes for providing a set of services within a certain domain. An application framework is a collection of classes that provide the services necessary for creating applications for a specific operating system.

Application Frameworks and MFC

And that brings us to the Microsoft Foundation Classes—an application framework specifically tailored for creating applications for Microsoft’s Windows operating system. And boy do we need one!

The Microsoft Software Development Kit (SDK) documentation presents the entire Windows Application Programming Interface (API) alphabetically. Microsoft documents the API fairly well, and it’s easy to find information (provided you first know *what* you’re looking for), but you can’t tell from the documentation which parts of the API are more important and how to fit the pieces together. Until recently, there was no “see also” addendum to the functions, and the relationships between the functions were hard to figure out. MFC helps by logically grouping the Windows API using the object-oriented principles of abstraction, encapsulation, inheritance, polymorphism, and modularity.

A Little History

In 1989, Microsoft established the application framework technology development group—a.k.a. the AFX group—to create C++ and object-oriented tools for Windows applications developers. (Legend has it that “AF” didn’t sound so great by itself, so they threw in the letter *X* to complete the acronym. The *X* doesn’t really mean anything.) Group members were experienced Microsoft developers culled from a pool with widely varied backgrounds. Some came from the group that designed and implemented the common components shared by Microsoft applications, others came from

the original Windows and Presentation Manager teams, and still others came from the applications division. The group's charter was to "utilize the latest in object-oriented technology to provide tools and libraries for developers writing the most advanced GUI applications on the market."

Unfortunately, the first prototype application framework turned out to be completely unrelated to Windows. After a year of work, the AFX group ended up with an application framework that contained its own complete windowing system, its own graphics subsystem, a custom control architecture, an object database, a generalized container hierarchy, and a garbage collection scheme! But when the group tried to write applications using the framework, they decided that it was far too complex and far too different from Windows itself. So, as Sinofsky reported, they scrapped the prototype and revised their charter to read: "AFX will deliver the power of object-oriented solutions to programmers to enable them to build world-class Windows-based applications in C++." (This was about the same time that Windows 3.0, the first workable version of Windows, was released.) The new focus was on creating a simple, elegant, and technologically feasible framework for Windows developers using C++, which provides a path to future Windows environments.

Design Goals for MFC

The AFX group had five main priorities for their new framework:

- Deliver a real-world application developed using C++ and object-oriented techniques.
- Deliver an application framework that simplifies Windows development for both new and experienced programmers.
- Allow developers to leverage their existing knowledge of Windows.
- Establish a foundation for Windows developers to build large-scale applications that utilize the latest Windows enhancements, such as Object Linking and Embedding (OLE) and Open Database Connectivity (ODBC).
- Keep the framework small and fast.

Delivering a Real-World Application

While trying to deliver a real-world application developed using C++ and object-oriented development techniques, the AFX group decided that instead of defining new programming concepts, they should extend the object-oriented programming model already used by Windows and express their concepts in standard C++ idioms. C++ is a very complex language, and the AFX group anticipated (correctly) that not everyone using MFC would be a C++ guru. So instead of using the entire

C++ language, they defined a subset of C++ that they would use to implement MFC. However, they placed no restrictions on code a programmer could write: you can use any feature of C++ you want to.

Simplifying the Windows API

To achieve the second goal of delivering an application framework that simplifies Windows development, the AFX team grouped the API functions into logical units. In doing this, they took advantage of C++'s classes and inheritance to make development conceptually easier. For example, all the functions related to a device context appear in one class, all the functions related to a generic window appear in another, and so on.

They also used the strong typing and data-encapsulation features of C++ to make development mechanically easier. For example, every Windows message uses two parameters: a WPARAM and an LPARAM. However, the actual values stored in a message may be any number of things: pointers, POINT structures, window handles, and so on. To retrieve message parameters, the C/SDK programmer needs to decode the parameters into the proper type of values. This is generally done through typecasting, which completely circumvents the benefits of function prototyping and strict data typing and introduces a possibility of error. MFC encapsulates the parameter packing and unpacking for you, so as an MFC programmer, you deal with parameters in their native types, and you can use the compiler to verify that the proper types of values are passed to functions.

Finally, MFC safely hides away the messy details of Windows programming, such as registering window classes, window and dialog procedures, and message routing.

Using the Knowledge You Already Have

Instead of rewriting Windows (as the first prototype produced by the AFX group did), the group used existing Windows features whenever and wherever possible. To minimize the relearning curve for experienced developers, they based the class hierarchy and naming conventions on the underlying API naming conventions. Generally, the result is that converting a standard program written using the SDK to MFC is fairly simple. In fact, you may not even need to convert the code. MFC was designed so that low-level C code could be reused within an MFC-based application. Don't throw away your copy of *Programming Windows*. The existing literature on Windows programming remains useful to you even when using MFC!

A Firm Foundation

The Windows API is like a snowball rolling down a hill, getting bigger and bigger as it continues to collect new material. Microsoft keeps adding new APIs to Windows,

including OLE 2.0 and ODBC. The advent of Windows NT brought with it a new 32-bit API with still more features. Although the initial framework doesn't support every new Windows API, MFC was designed so that it could be readily extended. Each new API introduces a new learning curve to developers (*arrgh!*), but MFC eases the burden by encapsulating the details of an API in a set of classes. For example, the OLE classes in MFC reduce by thousands the number of lines of code required to OLE-enable your application.

Keeping It Small and Fast

The AFX group wanted the framework to be small and fast. Their original goal for MFC was to have a 20K memory price tag for using the framework and for applications developed with MFC to be no slower than an equivalent C/SDK application. This constraint has a significant impact on both the overall design and the implementation of MFC. Class libraries with high levels of abstraction and many virtual functions tend to produce large, slow applications. To keep the speed up and the size down, the AFX group invented other mechanisms to handle Windows messages. MFC keeps the flexibility without the overhead of tons of virtual functions.

The First Release

In 1992, Microsoft released MFC version 1.0 with both their C/C++ version 7.0 and the SDK that was shipped with Windows NT. MFC 1.0 included more than 60 C++ classes for Windows application development, as well as general-purpose classes for time, strings, collections, files, persistent storage, memory management, exceptions, and diagnostics. The classes were optimized to execute very quickly with minimal memory requirements. Instead of being a whole new, high-level abstraction for Windows development, MFC 1.0 provided a thin, efficient C++ transformation of the Windows API.

But That Wasn't Enough . . .

MFC 1.0 received critical acclaim, and was recognized by some as the standard application framework for Windows. However, nothing's perfect, and the AFX group received a great deal of constructive feedback on MFC 1.0 from real-world developers desiring improvements. From the hundreds of suggestions and feature requests they received, the group determined that two new main features should be added for the next release, MFC 2.0: (1) high-level architecture support and (2) canned components.

The AFX group added a high-level architecture and a set of canned components to simplify Windows development even more than MFC 1.0 did. This addition allowed

developers easily to build Windows applications that followed the user-interface guidelines presented in Microsoft's application interface design guide, and that contained the same popular features offered by big-name applications, such as toolbars, status bars, print preview, and context-sensitive help. MFC's architecture provided hooks for the developer to customize and extend the generic application. However, MFC 2.0 didn't require the developer to use this high-level abstraction. The developer could still access the framework at lower levels, much like MFC 1.0, or even access the Windows API directly.

When designing the high-level architecture, the AFX group kept in mind the future direction of Windows itself. Windows continues to move toward an object-oriented operating system based on OLE 2.0. Although OLE 2.0 had not yet been released when MFC 2.0 came out, the architecture is designed so that OLE 2.0 support is a natural extension of the existing architecture.

The AFX group also considered some additional constraints: compatibility and portability. Applications written using MFC 1.0 were compatible with MFC 2.0. Developers didn't want to rewrite their MFC 1.0 applications just to use some new features of MFC 2.0. In addition, applications written using MFC were portable between application target platforms. Porting an application written using MFC 2.0 for Windows 3.1 to run under Windows NT or Win32s is as simple as recompiling the application using the 32-bit version of MFC 2.0.

Microsoft released Visual C++ version 1.0 for Windows with MFC version 2.0 in April 1993. Following the release of Windows NT, Visual C++ version 1.0 for NT shipped with MFC version 2.1. The two versions had essentially the same feature set, although the underlying implementations were slightly different. MFC 2.0 encompassed most of the base Windows API, plus OLE 1.0. It contained nearly 60,000 lines of standard C++ source code in over 100 classes! Though the general-purpose and Windows classes present in MFC 1.0 were enhanced, the primary features of this release were the new application architecture and the high-level abstractions. The application architecture classes provided standard implementations of common Windows features, such as documents and views, printing, and command processing. The high-level abstractions provided a set of prebuilt components for common user-interface elements, such as toolbars and status bars.

Where Are We Now?

In late 1993, Microsoft released Visual C++ version 1.5 with MFC version 2.5. MFC 2.5 built on the high-level architecture introduced in MFC 2.0 to add support for OLE 2.0 and ODBC. In late 1994, Microsoft released MFC version 3.0, in which the main improvement was thread safety. It is now 1996. Early in 1995, Microsoft added simple Messaging Application Programming Interface (MAPI) and WinSock

support to version 3.1. Then in late 1995, Microsoft released VC++ 4.0 (the development environment) and MFC 4.0 (the framework) simultaneously, greatly enhancing the development environment, with a central focus on the theme of reusability.

At this point, Microsoft has added another very compelling reason for using MFC: they would like to see MFC become the C++ interface for Windows programming. You'll see more MFC/C++ code in the future simply because it's sanctioned by Microsoft. So, it behooves you to learn as much about MFC as you can. This book is a great place to start.

A Grand Tour of MFC

The first stop on the journey into the depths of MFC is to take a broad look at the classes that compose it and how they work together. MFC is big. It has to be—it encapsulates most of the Windows API and provides a robust application framework. Still, it can be broken down into manageable chunks. At the outset, Microsoft grouped the classes into four general categories:

- General-purpose classes
- Windows API classes
- Application framework classes
- High-level abstractions

The general-purpose classes provide things like a string-handling class, collection classes, and exception classes. The Windows API classes wrap up the Windows API. Here you'll find window classes, dialog box classes, device context classes, and so on. The application framework classes handle the large pieces of a whole application. The message-pumping logic, as well as printing, on-line help, and MFC's document/view architecture, are encapsulated here. The high-level abstractions include nonbasic features such as toolbars, splitter windows, and status lines. Finally, MFC provides support for several operating system extensions, including OLE, ODBC, Simple MAPI, and WinSock. The rest of this chapter provides a closer view of each of these groups.

General-Purpose Classes

When you think of an application framework for Windows, the first classes you probably think of are window classes, application classes, and control classes. But MFC contains a smorgasbord of classes that have nothing to do with the visual elements

or the obvious structural elements of an application framework. Still, the general-purpose classes are important to MFC, and without them MFC wouldn't work nearly as well as it does (and in some cases, not at all). In essence, they help glue the framework together. The general-purpose classes include CObject, the exception-handling classes, the collection classes, dynamic strings, file classes, date and time classes, and a few assorted miscellaneous classes.

CObject: The Mother of (Almost) All Classes

Many class libraries provide one or two abstract classes from which most of the other classes in the library are derived. This is MFC's strategy, as well. MFC's root class is called CObject. Classes derived from CObject inherit some useful capabilities, including run-time type identification for the derived class, serialization (that is, persistence), diagnostic functions, and support for dynamic object creation. These features are optional, and may be enabled or disabled when a new derived class is defined.

To supply these benefits, a CObject employs the services of several other MFC classes:

- CArchive handles object persistence in conjunction with CObject.
- CDumpContext is used by the diagnostic functions to output information about an object.
- CRuntimeClass is associated with every class derived from CObject and contains information about the class.

Exception-Handling Classes

Exceptions are abnormal conditions that occur beyond the control of your program, including things like low memory or I/O errors. MFC offers a way to handle exceptions by providing a set of exception-handling classes. Many MFC functions will throw an exception, and you can protect your program by surrounding calls to MFC functions that throw exceptions with an exception handler.

MFC's predefined exception-handling classes include these:

- CException—The base class for exceptions.
- CArchiveException—Handles exceptions that occur during archiving.
- CFileException—Handles exceptions that occur during file operations.
- CMemoryException—Handles memory exceptions.
- CNotSupportedException—Handles exceptions thrown by a program trying to use an unsupported feature.
- COleException—Handles exceptions for both OLE clients and servers.
- CDBException—Handles exceptions thrown during data access processing.

- CResourceException—Handles exceptions thrown by a failure to read a Windows resource.
- CUserException—Handles an application-specific exception.

Memory Diagnostics

The CMemoryState structure provides a handy way of tracking memory leaks in your program by setting up memory checkpoints, and it supplies member functions for controlling and querying the memory checkpoints.

Collection Classes

A collection class manages a group of objects. All MFC's collection classes contain methods to add and delete elements from the collection, to retrieve elements from the collection, and to iterate over the entire collection. All that code you used to write to dynamically resize arrays and to maintain linked lists can now be handled by MFC's collection classes, which include the following:

Collections Implemented as Arrays

- CByteArray—An array of bytes.
- CWordArray—An array of word values.
- CDWordArray—An array of double word values.
- CPtrArray—An array of pointers.
- COBArray—An array of objects derived from CObject.
- CStringArray—An array of CString objects.
- CUIntArray—An array of unsigned integers.

Collections Implemented as Linked Lists

- COBList—A linked list of objects derived from CObject.
- CPtrList—A linked list of pointers.
- CStringList—A linked list of CString objects.

MFC also supplies a set of collection classes known as maps. A map lets you take two objects and pair them, making it easy to look up one object as long as you have the other. For instance, if you wanted to create a dictionary of digits and their spellings, you could use the word-to-string map to pair a number with its spelling. MFC supports the following maps:

- CMapPtrToWord—Maps a pointer to a word.
- CMapPtrToPtr—Maps two pointers.

- CMapStringToOb—Maps a string to an object derived from CObject.
- CMapStringToPtr—Maps a string to a pointer.
- CMapStringToString—Maps two strings.
- CMapWordToOb—Maps a word to an object derived from CObject.
- CMapWordToPtr—Maps a word to a pointer.

Dynamic Strings

If you are continually frustrated with C's arcane character string functions, use MFC's dynamic strings. Creating strings with the CString class is a snap. MFC's CString class provides basic operations such as concatenation, comparison, and assignments.

File Classes

MFC's file classes encapsulate file I/O. CFile handles standard binary file I/O, and CStdioFile handles buffered I/O files (such as text files). Finally, CMemFile handles memory files. These are binary files that behave as though they were on disk, but they actually reside in memory.

Time and Time Span Classes

Two classes, CTime and CTimeSpan, make it easy to work with time values in your programs. The CTime class represents the absolute time and provides member functions to get the current time, format a specific time into a string, and extract specific elements (such as minute, second, month, and year). CTime also provides operators for adding, subtracting, comparing, and assigning times.

The CTimeSpan represents time in seconds and is useful for showing an elapsed time. This class provides methods to extract the elements from it (days, hours, seconds, total days, total hours, and total seconds) as well as addition, subtraction, comparison, and assignment operators.

Some Other Miscellaneous Classes

Wrapping up the general-purpose classes are a few Windows structures implemented as objects:

- CPoint—Encapsulates the POINT structure.
- CSize—Encapsulates the SIZE structure.
- CRect—Encapsulates the RECT structure.

Windows API Classes

One of the most important things MFC does is wrap up the Windows API, hiding all that grunge and boilerplate code that you may not want to know about. The Windows API classes also encapsulate certain Windows objects such as device contexts. This is no easy feat, but MFC did it. Here's a rundown of the Windows API classes within MFC. Some classes are woven deeply into the high-level architecture, but others can be used independently (for instance, you can use the control classes in a regular C/SDK program).

Your Application: CCmdTarget, CCmdUI, CWinThread, and the CWinApp Classes

In a standard C/SDK application, you write a `WinMain()` function that picks messages out of a message queue and passes them on to a window procedure (also called a message handler). The message handler you write is a (usually huge) case statement in which you trap the messages you want, handle them, and let the other messages go by.

MFC's solution to this mess is message mapping, and it is implemented by the `CCmdTarget` and `CWnd` classes. A message map is the mechanism by which events are mapped to a class's method for handling that event. Any object that is derived from the `CCmdTarget` class can have a message map associated with it, which routes commands to the `CCmdTarget`-derived class. The class handles the messages using various member functions.

The `CCmdUI` class provides a way for updating the state of a user-interface object, such as a menu or a checkbox control. For example, instead of calling the `EnableMenuItem()` API function, the `CCmdUI` class lets you use messages to cause the command target object to update the menu status automatically. Before showing the menu items when a menu is clicked, MFC sends command update messages to the command targets in an application. If there is an entry in a command target's message map for the update message, MFC passes a pointer to the `CCmdUI` object representing a menu item, which the command target updates.

`CWinThread` represents a thread of execution within an MFC program. Though earlier versions of MFC weren't thread safe, versions 3.0 and later are. This is accomplished by supporting two different kinds of threads: worker threads (real, preemptive Win32 threads) and user-interface threads (which are really nothing more than a standard message pump using `GetMessage()`..`DispatchMessage()`).

In standard C/SDK, the heart of an application is the message loop, which looks like this:

```
while (GetMessage((LPMMSG)&msg, NULL, 0, 0)) {
    /* Check for menu accelerator message */
    if (!TranslateAccelerator(hwndMain, hAccel, &msg)) {
```

```

    TranslateMessage( (LPMMSG)&msg );
    DispatchMessage( (LPMMSG)&msg );
}
}

```

The CWinThread encapsulates this behavior. That is why you won't find a WinMain() function within your MFC code. The functionality that composes a standard WinMain() function is tucked neatly away inside the CWinThread class and the libraries that make up MFC.

CWinApp, which is derived from CWinThread, represents the standard Windows application. CWinApp has overrideable functions you can use to initialize your app and perform termination cleanup to suit your own needs. Because CWinApp is derived from CWinThread, it has a Run() method that kicks the application into high gear and executes it.

Synchronization Object Classes

MFC 4.0 introduced a set of classes that simplify programming multi-threaded applications by providing a variety of wrappers around the different Win32 synchronization methods. These classes are the following:

- CSyncObject—The base class of the synchronization object classes.
- CCriticalSection—A synchronization class that allows only one thread within a single process to access an object.
- CSemaphore—A synchronization class that allows between one and a specified maximum number of simultaneous accesses to an object.
- CMutex—A synchronization class that allows only one thread within any number of processes to access an object.
- CEvent—A synchronization class that notifies an application when an event has occurred.
- CSingleLock—Used in member functions of thread-safe classes to lock on one synchronization object.
- CMultiLock—Used in member functions of thread-safe classes to lock on one or more synchronization objects from an array of synchronization objects.

Related Classes

Two other classes that are used application-wide are these:

- CCommandLineInfo—Parses the command line with which your program was started.
- CWaitCursor—Puts a wait cursor on the screen. Used during lengthy operations.

CWnd: The Mother of All Windows

CWnd is the class from which all other windows (such as dialog boxes, controls, and frame windows) are derived. Because CWnd is derived from CCmdTarget, it can receive and handle messages. Inside CWnd, you'll find the specific window's handle, most of the common API functions that operate on a window (such as ShowWindow(), MessageBox(), and MoveWindow()), as well as some new ones that are specific to MFC.

Frame Windows

The main frame window is usually the first window in an application to receive messages. CFrameWnd serves as the base class for a Single Document Interface (SDI) application's main window. CMDIFrameWnd provides a main frame window for Multiple Document Interface (MDI) applications, and CMDIChildWnd provides child windows for an MDI application.

CWnd Derivatives: Dialogs and Controls

CWnd serves as a breeding ground for all of Windows' visual interface objects. CDialog, and all the controls (CButton, CListBox, CEdit, and so on), are derived from CWnd, so they have all the same functionality as regular windows, as well as special functionality specific to themselves. For instance, in addition to the standard window methods, the CListBox class has methods for filling a list box, getting a selection, setting a selection, and so on.

Dialogs

MFC's dialog boxes are supported by the CDialog class, which lets you create either modal or modeless dialog boxes. In addition to being able to create your own dialogs, MFC provides a set of classes that encapsulates the Windows common dialogs so that you don't have to cough them up yourself. The common dialog boxes also add a similar look and feel across applications. They are the following:

- CFileDialog—Selects a file from a directory.
- CColorDialog—Selects a specific color.
- CFontDialog—Selects a font.
- CPrintDialog—Handles printer setup and printing.
- CFindReplaceDialog—Selects text to search and replace.

Dialog Data Exchange and Validation

When writing dialog boxes in a regular C/SDK program, you have to write code to both initialize your controls and collect the data from the controls when the OK

button is pressed. MFC has provided a means of initializing dialog controls and automatically extracting the data from them at the end of the dialog. Dialog data exchange and validation (DDX/DDV) is implemented through the `CDataExchange` class.

Property Sheets

Using Visual C++, you've probably noticed property sheets, or tabbed dialogs. They are useful for reducing the number of dialogs in your application by layering them all in one dialog. MFC supports property sheets through two classes: `CPropertySheet` and `CPropertyPage`.

Controls

MFC wraps each of the basic Windows controls into their own class. All controls are derived from the `CWnd` object. They are the following:

- `CButton` and `CBitmapButton`—A standard Windows pushbutton and custom pushbutton with a bitmap.
- `CCComboBox`—A standard Windows combo box.
- `CEdit`—A standard Windows edit control.
- `CScrollBar`—A standard Windows scroll bar.
- `CListBox`—A standard Windows list box.
- `CCheckListBox`—A special list box that contains a list of buttons.
- `CStatic`—A standard Windows static control.
- `CVBControl`—A Visual Basic control (16-bit MFC only).

In addition to these standard controls, MFC makes it easy to create owner-draw control by including owner-draw methods with the `CWnd` class.

MFC also includes control classes that wrap the new Windows common controls. These classes are the following:

- `CAmputeCtrl`—A control that plays animations.
- `CDragListBox`—A `CListBox` derivative that lets you drag and drop items in a list box.
- `CHHeaderCtrl`—Used with a `CListCtrl` to display columnar information.
- `CHotKeyCtrl`—Provides an interface for getting key sequences from the user (Alt-Backspace-Delete).
- `CImageList`—A `CObject` derivative that maintains a collection of images for you.
- `CListCtrl`—Displays a graphical list (Explorer-like) of list items.
- `CProgressCtrl`—Displays a progress bar.

- CRichEditCtrl—A rich edit control that understands some basic RTF formatting and allows multiple fonts, colors, and so on.
- CSliderCtrl—A slider for choosing between a range of values.
- CSpinButtonCtrl—A spinner.
- CStatusBarCtrl—A status bar.
- CTabCtrl—The property sheet control.
- CToolBarCtrl—Implements a toolbar.
- CToolTipCtrl—Provides tool tips.
- CTreeCtrl—An Explorer-like tree control.

Menus

Windows menus are handled by the CMenu class. All the menuing functionality is encapsulated within CMenu, including the menu handle and methods to load, update, track, and destroy a menu. The CMenu class also supports owner-draw menus.

GDI Support and Drawing Objects

MFC supports the Windows Graphics Device Interface (GDI) with a variety of classes. The heart of the GDI is a device context (DC), which is represented in MFC by the CDC class. Creating a device context in MFC is as easy as creating an instance of a CDC. The device context class is handy because the actual device context is released when the device context destructor is called. This is good news for anyone who has forgotten to release his device contexts when he was done with them (we've never done it, of course). You'll find most of the device context-related API functions as member methods for the CDC. In addition to the CDC, MFC provides several additional device context classes for other specific uses:

- CPaintDC—Encapsulates calls to BeginPaint() and EndPaint() for use while handling a WM_PAINT message.
- CWindowDC—Encapsulates a device context associated with an entire window.
- CClientDC—Encapsulates a device context associated with a window's client area.
- CMetaFileDC—Encapsulates a device context for metafiles.

All the Windows drawing objects are also represented as classes within MFC. The CFont, CPen, CBrush, CBitmap, CPALETTE, and CRgn classes are all derived from the CGdiObject class, which provides the base functionality for all GDI objects.

Application Framework Classes

And now for the heavy stuff: the actual framework classes. These new features (not included in MFC 1.0) have greatly enhanced MFC as an application framework. Whereas MFC 1.0 helped organize the Windows API into manageable chunks, MFC 2.0 provides a systematic, robust way to create standard Windows applications using a real framework. The Application Framework Classes, covered next, were introduced in MFC 2.0. They are still a vital part of the framework and continue to be enhanced as MFC evolves.

Document/View Architecture

The document/view architecture is probably the most prominent difference between versions 1.0 and 2.0 of MFC. The idea behind the document/view architecture is to separate actual data from representations of that data. In other words, for one set of data, you can generate multiple views of it. A spreadsheet is a good example. A single spreadsheet can contain rows and columns of numbers. Yet from that data you can generate a number of graphs. When you change a number in the spreadsheet—voilà—it gets changed in the graphs instantly. MFC provides several classes that implement the document/view architecture. These include the following:

- **CDocTemplate**, **CSingleDocTemplate**, and **CMultiDocTemplate**

The document template is the glue that holds together a document and its views. It is the liaison between documents, views, and frame windows. A CSingleDocTemplate supports one document at a time, whereas a CMultiDocTemplate supports multiple documents simultaneously. Both are derived from the CDocTemplate class.

- **CDocument**

The CDocument class handles the data within your application. Typical documents might include spreadsheets, databases, and word processing text. Whatever you open using the File/Open command is usually your document (although a CDocument is not necessarily limited to disk-based data). When you derive a class from CDocument, you add variables to hold your data, code for reading and modifying your data, and code for writing your data back out to disk.

- **CView**

A CView represents the actual window you see on your screen. CView, which is derived from CWnd, is what you use to show your data to the outside world. The view renders your data on an output device such as a screen or printer. In a C/SDK program, you handle screen drawing when your application receives the WM_PAINT message, and the printing is handled somewhere else. MFC virtualizes this so that your CView's OnDraw() method is called whenever the view must be rendered. CView is set up so that the code that draws your

data on the screen is the same code that renders your image on the printer and during a print preview—it simply uses a different device context. The view is also a command target, so it can receive user input as well.

Context-Sensitive Help

Early on, context-sensitive help distinguished your application from others in the marketplace. This is no longer true. On-line help is becoming mandatory, so it's a good thing MFC includes lots of support for on-line context-sensitive help. MFC supports the following methods for getting on-line help:

- The Help menu—Calls for help from the menu.
- The F1 key—Gets help for the task at hand.
- Shift F1—Gets help on the next item the user clicks on.

High-Level Abstractions

Microsoft added a good many useful features to version 2.0 of MFC. Most of this is stuff you can't get à la carte—you get it only if you buy into the whole MFC document/view architecture.

Enhanced Views

You don't have to settle for a regular view of your data. MFC provides a number of canned enhancements to the CView class. These include scrolling views, form views, and edit views.

Scroll Views

If you have data that is too big for the screen, you can use the CScrollView. A CScrollView is like a viewport onto a page that you can move around by manipulating scroll bars. Alternatively, it can scale the view automatically to the size of the frame window that displays it, or you can implement the logic required to scroll the view yourself.

Form Views

One of the limitations of the Windows dialog manager is that it doesn't support many of the things that are required during true forms processing. A dialog box can have only as many controls as will fit on one screen (or 255 controls, whichever comes first). Also, dialog boxes don't support scrolling, printing, or multiple forms for the same data. This is a problem, because one of the fastest-growing uses for Windows is to host on-line data-entry applications. Big time data-entry operations (such as insurance companies) may need to fill in several pages of data at once. MFC resolves

these issues with form views. A CFormView is a scroll view melded with a dialog template. As with a regular dialog box, you can plant various controls (edit controls, list boxes, radio buttons, and so on). Because the CFormView is derived from a CScrollView, it has all the other inherent capabilities of a CScrollView, including scrolling, command handling, and document management through a document template.

Control Views

MFC 4.0 introduced several new CView derivatives that are basically controls with added CView features such as printing and print preview. The control views are these:

- **CEditView**—MFC's CEditView is like an edit control with added features. It does everything an edit control does, and more. The CEditView has all the characteristics of an edit control and can perform such operations as cut, copy, paste, undo, and find/replace. Because it's inherited from a CView, it shares all the functionality of a regular view. The CEditView makes it easy to stick a text editor into your application. (This view has been around since the 2.0 days. The others in this list are new with 4.0.)
- **CLListView**—CLListView lets you have a list of items in a view. It can be either a list of strings or anything you would like through an owner-drawn list.
- **CRichEditView**—A CView version of the popular new Rich Text Edit common control.
- **CTreeView**—Provides an Explorer-like tree in a view.

Splitter Windows

MFC provides a way to maintain several views of the same data in one window using separate, resizable panes. The CSplitterWnd class supports this interface. There are two kinds of splitter windows: static and dynamic. Static splitter windows have a predefined number of panes whose number and arrangement cannot be changed, though each pane can display a different type of view. On the other hand, the number and order of panes in a dynamic splitter window may vary, and each pane shows the same type of view.

Control Bars

MFC adds a whole new set of interface objects with the CControlBar class. Because they are derived from CCmdUI, they inherit the ability to receive command messages. Control bars come in three flavors: toolbars, status bars, and dialog bars.

Toolbars

CToolBar represents an application toolbar (though it goes by other, various names). A toolbar is a row of icons that provide an alternative to the regular pull-down menu.

MFC's CToolBar class makes it easy to include a toolbar in your application, similar to the ones found in Word for Windows or Visual C++ itself.

Status Bars

Along with a toolbar, most applications include a status bar somewhere on the screen. Usually the status bar displays information like the state of certain keys (Caps Lock, Scroll Lock, Num Lock, and so on) and a brief bit of text describing the current menu selection, although the status bar can be customized to display other information or even bitmaps.

Dialog Bars

In addition to toolbars and status bars, dialog bars are also new with Visual C++. They are like toolbars but contain regular dialog controls instead of bitmaps.

Operating System Extensions

Since version 2.5, MFC has supported a number of operating system extensions, including OLE, ODBC, Simple MAPI, and WinSock. Here's a look at some of the classes involved in supporting them.

OLE Support: OLE Documents

One of the more compelling reasons to use MFC is its support of OLE. OLE is becoming increasingly important, especially as Microsoft begins to emphasize its use within the operating system. MFC provides great support for the big-ticket OLE technologies: compound documents, automation, and OLE controls. MFC also wraps up OLE data transfer very nicely. Here's a rundown of the classes involved in creating documents that support OLE compound documents.

- **CDocItem**—The base class for MFC's COleClientItem and COleServerItem classes.
- **COleServerItem**—Represents the server's side of the connection to an embedded or linked OLE item.
- **COleClientItem**—Represents the container side of the connection to an embedded or linked OLE item.
- **COleDocument**—At the heart of MFC's compound document support. In addition to holding an application's native data, COleDocument maintains a list of CDocItem objects. Documents derived from this class can hold embedded and linked OLE items.
- **COleLinkingDoc**—Contains links to items that are embedded elsewhere.

- COleServerDoc—Used by applications on the server side of the compound document equation.
- COleIPFrameWnd—To qualify as a compound document server, an application has to contain two different frame windows: (1) a regular frame window (that is, the one shown on the screen when the application runs in native mode) and (2) a frame window that is used whenever the application is shown in place (that is, when the user invokes visual editing inside a compound document). The COleIPFrameWnd encapsulates that part of the compound document server functionality.

OLE Support: Class Factories

One requirement of OLE objects (if they are to be exposed from within servers) is that they have a class factory. A class factory resides inside an OLE server and creates instances of an OLE object on behalf of the server. The two classes that compose OLE class factory support are COleObjectFactory and COleTemplateServer.

- COleObjectFactory—Implements class factories for MFC applications that require class factories but are not document oriented.
- COleTemplateServer—Derived directly from COleObjectFactory. COleTemplateServer implements class factories for OLE-enabled MFC applications that are document oriented (such as compound document servers).

OLE Support: Automation

Automation is one of the most important features of OLE. Automation allows developers to open up their objects and make the properties and methods of their objects available to developers who use a more relaxed development environment than C++ (such as Visual Basic). MFC supports automation through the CCmdTarget class, using a mechanism called *dispatch maps*.

OLE Support: Uniform Data Transfer

OLE data transfer (a.k.a. Uniform Data Transfer) is accomplished through any object that implements the IDataObject interface. If you're programming using MFC, you don't have to implement IDataObject by yourself—all you need to do is use MFC's support for Uniform Data Transfer.

- COleDataSource—Every data transfer has two parts: a source and a destination. In an MFC program, data transfers are usually initiated using the COleDataSource class. This class is used for both Clipboard transfers and drag-and-drop transfers.

- COleDataObject—The other end of a data transfer, the destination, is usually represented using a COleDataObject.
- COleDropSource—Useful for customizing drag-and-drop operations.
- COleDropTarget—Whenever you are interested in creating a window that accepts data drops, you have to register that window's interest with MFC. COleDropTarget is the class to use for this purpose.

OLE Support: OLE Controls

OLE controls are quickly becoming one of the most important new technologies available through OLE. VBXs (Visual Basic controls) are on the way out, soon to be replaced by OLE controls. OLE controls implement most OLE technologies. They are automation objects that can also be embedded inside OLE documents. In addition to supporting an incoming automation interface, OLE controls can implement a number of outgoing interfaces that send events back to the host program. MFC has several new classes to support OLE controls:

- COleControl—Derived from CWnd, this is the base class for OLE controls. It extends basic CWnd functionality to deal with specific issues related to controls.
- COlePropertyPage—Derived from CDialog, this is the base class for OLE property pages. It's used for modifying a control's properties.
- COleControlModule—Derived from CWinApp, this is the base class for the dynamic link library (DLL) that holds the OLE controls. This is analogous to the CWinApp in an MFC program, which performs initialization and various tasks specific to OLE controls.
- COleObjectFactoryEx—This class is derived from and extends COleClassFactory. It does all the good things a regular class factory object should. In addition, COleObjectFactoryEx supports licensing. COleObjectFactoryEx was a separate class in MFC 3.0 but was merged into COleObjectFactory when the Control Developer Kit (CDK) extensions to MFC were merged into the core MFC.
- COleConnectionPoint—Derived from CCmdTarget, this class represents outgoing interfaces to other OLE objects. It is used in controls specifically for event firing and change notifications to the container.
- CPropExchange—This class is similar to MFC's CDataExchange class used for standard DDX/DDV. This class establishes the context of a property exchange and aids in exchanging properties between a control and the container.
- CFontHolder—This class encapsulates the Windows font object. It implements OLE's IFont interface and is used for the Font stock property.
- CPictureHolder—This class implements a “picture property.” It encapsulates a bitmap, an icon, or a metafile in a polymorphic way.

ODBC Support

MFC provided Open Database Connectivity support beginning with version 2.5. Here's a rundown of the ODBC classes:

- **CDatabase**—Encapsulates a connection to a data source, through which you can operate on the data source.
- **CRecordset**—Encapsulates a set of records selected from a data source. Record sets enable scrolling from record to record, updating records (adding, editing, and deleting records), qualifying the selection with a filter, sorting the selection, and parameterizing the selection with information obtained or calculated at run time.
- **CFieldExchange**—Supplies context information to support record field exchange (RFX), which exchanges data between the field data members and parameter data members of a record set object and the corresponding table columns on the data source.
- **CLongBinary**—Encapsulates a handle to storage for a binary large object (or BLOB), such as a bitmap. CLONGBinary objects are used to manage large data objects stored in database tables.
- **CRecordView**—Provides a form view directly connected to a record set object. The DDX mechanism exchanges data between the record set and the controls of the record view.

DAO Support

In MFC 4.0, Microsoft introduced another database access scheme called Data Access Objects (DAO). DAO is different from ODBC because it focuses on the Microsoft Jet (a.k.a. Access) database engine, instead of targeting all databases as ODBC does. The MFC classes for DAO support are these:

- **CDaoWorkspace**—Manages a named, password-protected database session.
- **CDaoDatabase**—Connects to a database through which you can operate on the data.
- **CDaoRecordset**—Represents a set of records selected from a data source.
- **CDaoRecordView**—Displays database records in controls.
- **CDaoQueryDef**—Represents a query definition, usually one saved in a database.
- **CDaoTableDef**—Represents the stored definition of a base table or an attached table.
- **CDaoException**—Represents an exception condition arising from the DAO classes.

- CDaoFieldExchange—Supports the DAO record field exchange routines used by the DAO database classes.

MAPI Support

There aren't any full MFC classes that support MAPI. Instead, it is embedded inside CDocument and allows you to send a document via electronic mail by calling the CDocument::OnUpdateFileSendMail() member function.

WinSock Support

There are two MFC classes that wrap the Windows Sockets interface. Class CAsyncSocket is a thin wrapper; CSocket supplies a higher-level abstraction for working with sockets.

Pen Windows Support

MFC supports Pen Windows with two new edit classes, CHEdit and CBEdit. CHEdit encapsulates the handwriting edit control in Microsoft Windows for Pen Computing. CBEdit is similar to CHEdit except that it displays boxes indicating where each letter should be entered. This gives the recognizer a better shot at figuring out what the user wrote. Pen support is limited to 16-bit platforms.

Conclusion

Now that you've had a quick tour of MFC, you can see that Microsoft has covered more than just the Windows API. They've also added the document/view architecture and several very useful high-level abstractions. The next stop is to compare a single program written two ways: one using C and the SDK and the other using C++ and MFC. In so doing, you'll see that MFC covers a lot of boilerplate code and adds support for things that you may not have even thought of using C and the SDK to do. The next chapter covers MFC's basic support for Windows applications, including WinMain(), the message pump, and MFC's message-routing mechanism.

Basic Windows Support

Take a breather after the last chapter. MFC is fairly extensive, isn't it? With over 200 classes, MFC offers a rich array of features. After seeing an overview of those classes, you may have an idea about which classes you'll want to use often and which classes you'll use only once in a while. You probably even saw some classes for which you might never find a use.

The point is, MFC is huge, and the only way to get through it is to divide it into digestible pieces. This chapter tackles MFC's basic Windows application support. No matter how you slice it, a Windows program is still a Windows program. It doesn't matter what language or framework you use. Regardless of whether a Windows program is written in C, C++, or Delphi, and regardless of whether it's created using an application framework like MFC or the Object Windows Library (OWL), all Windows programs do the same basic things—such as registering window classes and starting message loops. These tasks are just part of the Windows game. This chapter examines how MFC implements these fundamental tasks—the boilerplate code, if you will.

You've already had an overview of the classes and you should know the lay of the land. It's time to examine the most fundamental aspect of MFC's basic Windows application support: `WinMain()` and message handling. We'll start at the top and dig deeper into the framework for more detail as we go along. By looking at MFC this way, we'll revisit many classes several times. For example, besides simply encapsulating most of the window-oriented API functions, the `CWnd` class has to deal with the document/view architecture as well as support OLE 2.0. To accomplish this, `CWnd` has many aspects. We'll examine each side separately and then revisit `CWnd` while covering MFC's document/view architecture and OLE support in other chapters.

Let's start by examining how things happen in an MFC program versus how things happen in a C/SDK program. By taking a look at the way MFC actually encapsulates a Windows application, you'll see how MFC deals with the following issues:

- The application itself
- Windows
- Message handling
- The Graphics Device Interface (GDI)

To get a clear perspective on MFC, we'll compare two versions of a basic Windows program that does nothing more than watch the left mouse button for clicks. The first version uses the C/SDK approach. The second uses the C++/MFC approach.

MFC versus C/SDK

It has been said that only one Windows program has ever been written, and that the only way a Windows program distinguishes itself is by the way it handles messages. This is true in many respects. While there is a minimum set of requirements that a program must meet before it will run under Windows, the requirements are consistent from program to program. Every Windows application has two main components: the main application itself and at least one window that handles messages. In addition, much of this boilerplate code doesn't change from application to application. In fact, the *Guide to Programming* manual that comes with the SDK documents makes the following statement regarding the GENERIC.C program they provide as an example: "You can use Generic as a template to build your own applications. To do this, copy and rename the sources of an existing application, such as Generic; then change relevant function names, and insert new code."

Even Microsoft used to sanction the cut-and-paste (or "clone-and-code") method of Windows development. This type of development actually made some sense at one time. A minimal Windows program (one that does nothing but put a window up on the screen) written in C using the SDK requires about 80 lines of boilerplate code. Eighty lines of code is a lot, even if you drink a lot of coffee and type really fast. In addition, by typing that many lines from scratch, you can easily introduce a number of errors. However, since the boilerplate code is necessary, it's gotta get in there somehow. You might as well copy it from someplace where you know it works and then start adding to it.

However, working this way is time-consuming and inefficient. We're in OOP Land now, and things are done differently. It's no longer necessary to cut and paste

between whole applications. From now on you work with C++ classes— inheriting pieces of an application and changing only those parts you need to change.

All That (Boilerplate) Code

So why is all that boilerplate code necessary? Just what does a Windows application do that requires all that code? The quick answer is that an event-driven operating system imposes a great deal of overhead. There is a lot of setup that needs to take place for a Windows application, because Windows is an event-driven environment. A substantial amount of code is required for a Windows application to receive and process events.

Windows sits between the hardware and the applications it is hosting. Whenever anything happens that might be of interest to one of the applications, Windows informs the application. For example, imagine that the left mouse button has been pressed. Once Windows detects the event, there needs to be some way of letting the application know about the button press so the application can either process it in a meaningful way or ignore it.

Most of the overhead involved in Windows programming is a result of the machinery required to process messages this way. Consider the tasks that need to be accomplished:

- The Windows application needs to set up a message handler and plug it into Windows (via the RegisterClass() API function) so that Windows knows where to go to get messages handled for a specific application.
- Windows needs to keep track of specific instances of an application.
- The application asks Windows for the next event in the message queue, dispatches the message to the appropriate message handler, and then gets the next one.
- This activity continues until the application ends (for whatever reason).

With this in mind, it's easy to see that a large amount of code is required just to qualify the application as a Windows program. So, exactly what steps must an application take before it qualifies? There are several:

- The program must contain a WinMain() function. Just as any regular C program requires a main() function, every Windows program must contain a WinMain() function. WinMain() acts as the entry point for the program. It is through the WinMain() function that the application receives from Windows the information necessary for it to run. This information includes a handle to the current instance of the application, a handle to the last running instance of the application, any command line arguments, and how to show the window (minimized, maximized, normal, and so on).

- The application must register at least one window class to serve as the main window. What's a Windows program without a window? It is the window's job to present the user interface on the screen.
- The application must set up a message loop. If messages are the lifeblood of a Windows application, then the message loop is its heart. In Windows, it is the application's responsibility to set up the message pump. That is, the application needs to set up a loop to extract any messages.
- Most applications require some initialization and setup. This initialization usually takes two forms: application-specific and instance-specific initialization. The program needs to provide any necessary initialization steps.
- The application must provide a message handler. The function of the window procedure is to handle messages. At the very least, the window procedure must deal with the WM_DESTROY message (or the application will never end).

It would be nice if there was some way of encapsulating all that boilerplate code into a class (or a limited number of classes) so that you don't have to wear your fingers out on the keyboard. In fact, that is what MFC does.

Comparing the Code

A good way to see how an MFC program works is to compare a program that uses MFC with a standard C/SDK application. Listing 2-1 shows a minimal C/SDK application, much like many others you may have seen (or written). It's nothing special—it just initializes the application, puts up a main window, and starts the message loop, keeping an eye out for WM_LBUTTONDOWN messages and a WM_DESTROY message. However, it is a Windows program, and it demonstrates the basics of such a program. We can use it as a good starting place for investigating MFC.

Listing 2-1. The MINIMAL.C program, written with C/SDK

```

1 ****
2 ** Minimal.c- This is a minimal Windows program. It initializes
3 ** the program, starts a message loop, watches for WM_LBUTTONDOWN
4 ** messages, and waits for a WM_DESTROY message.
5 */
6 #include "windows.h"
7
8 HANDLE hInst; /* current instance */ 
9
10 long CALLBACK __export MainWndProc(HANDLE hWnd,
11                               UINT message,

```

```
12                               WPARAM wParam,
13                               LPARAM lParam) {
14
15     switch(message) {
16         case WM_LBUTTONDOWN:
17             MessageBox(hWnd, "Left mouse button clicked...", NULL, MB_OK);
18             break;
19         case WM_DESTROY:
20             PostQuitMessage(0);
21             break;
22
23         default: /* Passes it on if unprocessed */
24             return (DefWindowProc(hWnd, message, wParam, lParam));
25     }
26     return (NULL);
27 }
28
29 BOOL InitApplication(HANDLE hInstance) {
30     WNDCLASS wc;
31
32     wc.style = NULL;           /* Class style(s).          */
33     wc.lpfnWndProc = MainWndProc; /* Function to receive messages for */
34                                         /* windows of this class.    */
35     wc.cbClsExtra = 0;        /* No per-class extra data. */
36     wc.cbWndExtra = 0;        /* No per-window extra data. */
37     wc.hInstance = hInstance; /* Application that owns the class. */
38     wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
39     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
40     wc.hbrBackground = GetStockObject(WHITE_BRUSH);
41     wc.lpszMenuName = NULL;   /* Name of menu */
42     wc.lpszClassName = "MinimalWClass"; /* Name of window class */
43
44     return (RegisterClass(&wc));
45 }
46
47 BOOL InitInstance(HANDLE hInstance, int nCmdShow) {
48     HWND hWnd;                /* Main window handle. */
49
50     hInst = hInstance;
51
52     hWnd = CreateWindow(
53         "MinimalWClass",      /* Window Class          */
54         "Minimal",            /* Caption               */
55         WS_OVERLAPPEDWINDOW, /* Window style.         */
56         CW_USEDEFAULT,        /* Default horizontal pos. */
57         CW_USEDEFAULT,        /* Default vertical pos. */
58         CW_USEDEFAULT,        /* Default width.        */
59         CW_USEDEFAULT);       /* Default height.       */
```

34 • BASIC WINDOWS SUPPORT

```
60     NULL,           /* No parent.          */
61     NULL,           /* Use the window class menu.   */
62     hInstance,       /* This instance owns the window.*/
63     NULL);          /* Not needed.         */
64
65     if (!hWnd)
66         return (FALSE);
67
68
69     /* Show window, update window */
70
71     ShowWindow(hWnd, nCmdShow);
72     UpdateWindow(hWnd);
73     return (TRUE);
74 }
75
76 int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
77                      LPSTR lpCmdLine, int nCmdShow) {
78     MSG msg;           /* message */
79
80     if (!hPrevInstance) { /* First instance of app? */
81         if (!InitApplication(hInstance)) /* Shared stuff */
82             return (FALSE); /* cannot initialize */
83     }
84
85     if (!InitInstance(hInstance, nCmdShow))
86         return (FALSE);
87
88     while (GetMessage(&msg, NULL, NULL, NULL)) {
89         TranslateMessage(&msg);
90         DispatchMessage(&msg);
91     }
92
93     return (msg.wParam); /* Returns the value from PostQuitMessage */
94 }
```

Whew! That's a lot of code! The same program using MFC appears in Listing 2-2.

Listing 2-2. The MINIMAL.CPP program, written with C++/MFC

```
1 #include <afxwin.h>
2
3 // Define an application class derived from CWinApp
4 class CGenericApp : public CWinApp {
5 public:
6     virtual BOOL InitInstance();
7 };
8
9 class CGenericWindow : public CFrameWnd {
```

```

10 public:
11     CGenericWindow() {
12         Create(NULL, "Generic");
13     }
14
15     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
16     DECLARE_MESSAGE_MAP()
17 };
18
19 BEGIN_MESSAGE_MAP(CGenericWindow, CFrameWnd)
20     ON_WM_LBUTTONDOWN()
21 END_MESSAGE_MAP()
22
23 void CGenericWindow::OnLButtonDown(UINT nFlags, CPoint point) {
24     MessageBox( "Left mouse button pressed...", NULL, MB_OK );
25 }
26
27 BOOL CGenericApp::InitInstance() {
28     m_pMainWnd = new CGenericWindow();
29     m_pMainWnd->ShowWindow(m_nCmdShow);
30     m_pMainWnd->UpdateWindow();
31     return TRUE;
32 }
33
34 CGenericApp GenericApp;

```

That's a little better, wouldn't you say? By getting rid of the boilerplate code, we reduced the size of the source code by more than 60%!

So, what's happening in each of these applications? First, examine the C/SDK example. You can clearly see everything that is happening—nothing is left to the imagination. The program obviously fulfills the necessary requirements for being a Windows application. There's a WinMain() function. A window class is registered in InitInstance(). The main window is created and shown. Finally, there is a window procedure for the main window (it's called MainWndProc()) that handles the left mouse button clicks. No surprises here.

Now check out the MFC example. It looks as if it's missing some things. For one, the WinMain() function is gone, and it doesn't look as if any window classes are being registered. Furthermore, there's no message-handling function, and it doesn't even look as if a message loop is being created. Yet when you run it, it does exactly the same thing as the C/SDK version. It doesn't seem possible. Where did all the code go? Well, the code is still in there—it's just playing hide-and-seek.

Writing a Windows program using MFC (or another application framework, such as Borland's OWL) is in many ways an act of faith, especially if you have a strong background in the ways of C and the SDK. Using C and the SDK, everything is

clearly in view, and it's usually fairly obvious what's going on in the program. The downside is that you end up cranking out a lot of C code. The beauty of the MFC approach is that it radically cuts down on the amount of code that you have to write. In fact, all you really need to do to get a minimal Windows program to run is to instantiate a CWinApp object and create a main window object. Simply instantiating the CWinApp causes the program to run "automagically." The difficult thing is that you have to take it on faith that everything is in there and that it will work as advertised. However, when you do take a peek under the hood, all the necessary machinery is there—if you look for it.

Of course, you may have had to deal with this type of paradigm shift before when moving from DOS programming to Windows programming. The idea is that the operating system is becoming a part of your application (in a manner of speaking). Think about it: When you programmed in DOS, you more or less had complete reign over the system. The operating system was simply there to provide very rudimentary services for you. DOS was your slave (although some will contend that a whole generation of programmers became slaves to DOS). But in Windows programming, Windows and the application need to become much more cooperative. An application becomes involved in a cybernetic loop with Windows, with each part passing messages back and forth to the other.

Moving to MFC requires a similar paradigm shift. The code to handle 99% of a generic application is there. However, rather than set up window procedures to trap events, parts of the application framework become notified as events happen. That is, the message traps are already installed. All you have to do is decide which parts of the framework to override.

Now let's take a closer look at the MFC application and figure out where everything went.

The WinMain() Function

To be able to run under Windows, a program must supply a WinMain() function as an entry point. This is where most programs perform application- and instance-specific initialization, as well as start up the application's message loop.

In MINIMAL.C, you can see this in lines 76 through 94. If this is the first running instance of the application, WinMain() calls InitApplication(), which registers a window class called MinimalWClass. WinMain() then calls InitInstance(), which creates and shows a window cast from the MinimalWClass mold. Finally, WinMain() goes on to create the application message loop, which fetches messages from the message queue and dispatches them to the appropriate window until a WM_QUIT message is encountered.

Now look closely at MINIMAL.CPP. There's no WinMain() function. Where did it go? WinMain() is there—it's part of the application framework; you link it in when you link in the framework code. MFC's WinMain() function does all the things that a good WinMain() should do, including initializing the application and starting up the message loop.

Initializing a Particular Instance of the Application

There are some initializations that have to be done for every running instance of the application. Usually, this means creating and showing the main window.

In MINIMAL.C, initialization happens between lines 47 through 74 in the function InitInstance() (which was called by WinMain()). InitInstance() preserves the handle to the current instance of the program in a global variable, hInst. In more involved programs, this value is needed for loading resources. Because the only time Windows gives your application the handle to the current instance is at the beginning of the program (during WinMain()), now is the time to save it if you're going to need it. InitInstance() then creates the main window and shows it on the screen in the manner specified by the nCmdShow parameter that was passed in on the command line.

In MINIMAL.CPP, instance-specific initialization happens between lines 27 and 32. Although you can't see it here, CGenericApp calls its member function InitInstance() from within WinMain(). InitInstance() creates a new main window and tells it to show itself.

The Message Loop

The message loop is where the application picks up and dispatches each message. In MINIMAL.C, the message loop is covered between lines 88 and 91. There are no surprises here: the loop keeps calling GetMessage() and DispatchMessage() until a WM_QUIT message is encountered.

In MINIMAL.CPP, the message loop is again part of the framework. It gets started when CWinApp's Run() member function is called as the last step of WinMain(). MFC's message handling provides some extras, such as performing idle processing when there are no messages in the queue, and supplying a PreTranslateMessage() function for intercepting special messages. However, at the very lowest level, MFC pumps and dispatches messages through the system in the conventional fashion, calling GetMessage(), TranslateMessage(), and DispatchMessage(). The messages are then handled by MFC's command-routing architecture.

Message Handling

Every Windows class within a Windows application requires a message-handling procedure. It is through the message handler that a window derives its specific behavior. Whenever Windows detects an event that is pertinent to a specific window, it generates a message and calls the window's message handler with information about that event. For example, in MINIMAL.C, the main window's behavior includes responding to WM_LBUTTONDOWN and WM_DESTROY messages. Notice that the message handler (lines 10–27) does nothing more than filter out these messages using a switch statement, calling DefWindowProc() if the message is anything other than a WM_LBUTTONDOWN or a WM_DESTROY message. Other applications might have to respond to menu commands or mouse movement commands, in which case the switch statement grows to accommodate the new messages.

Notice that the MINIMAL.CPP source code is completely devoid of a window procedure. In its place are two things:

1. A member function for the main window, called OnLButtonDown (lines 23–25).
2. A message map (line 16 and lines 19–21). Instead of assigning a specific message handler to a window (as a C/SDK program would), an MFC program uses message maps to get commands and messages to a command target. (For example, a CWnd is a command target: it is the target of commands and messages.) Message maps associate specific commands and messages with member functions that handle the message.

MFC wouldn't work if there wasn't a window procedure. In fact, there is a window procedure associated with CWnd. If you dig around in APPINIT.CPP (MFC source code is included with Visual C++), you'll find four window classes registered. Even though there are four different window classes, they all use the same window procedure (called DefWindowProc()). That doesn't sound promising, does it? Well, if you root around a little more, you'll see that MFC has another window procedure, called AfxWndProc(). Though this function isn't installed directly through window class registration, AfxWndProc() is wired in eventually using Windows hooks (we'll see how shortly). The CWnd class has a WindowProc() member function. AfxWndProc() ends up calling CWinApp's m_pMainWnd->WindowProc(), which then routes the message to the right command target.

So although there is not an explicit window procedure for the window class, the messages are effectively handled by the command-routing and message-dispatch architecture. MFC handles WM_COMMAND through the command-routing mechanism, which gets a WM_COMMAND message to the appropriate command target. MFC's message-dispatching mechanism gets a window message to the appropriate window, where it's handled by the appropriate member function.

Basic MFC Application Components

Underneath the hood, all Windows applications are very similar to the previous C/SDK example as far as the fundamental structure is concerned. Again, a Windows app contains at least two distinct parts: a message pump and a window procedure. MFC supports this basic Windows program structure by segregating Windows applications into two main pieces: a class representing the application and a class representing a window. This is a logical way to treat things—just thumb through the reams of C/SDK source code available today. C/SDK programs all contain these two distinct parts: (1) an application-specific part that handles initialization, creating one or more windows, and sustaining a GetMessage()..DispatchMessage() loop and (2) a window-specific part that takes care of drawing on the window, message handling, and so on. In MFC, these two main components are embodied within the `CWinApp` and the `CWnd` classes, respectively.

Enough about the basics. Let's take a closer look at the structure of an MFC program.

CWinApp: The Application Object

Of course `CWinApp` is quite a large class—it has a lot to do. Listing 2-3 is an abridged version of the class (the whole definition is in `AFXWIN.H`).

Listing 2-3. The CWinApp pseudocode, from AFXWIN.H

```
class CWinApp : public CWinThread
{
public:

    // Constructor
    CWinApp(LPCTSTR lpszAppName = NULL); // app name defaults to EXE name

    // Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
    int m_nCmdShow;
    LPCTSTR m_pszAppName; // human readable name
    // advanced attributes omitted

public: // set in constructor to override default
    LPCTSTR m_pszExeName; // executable name (no spaces)
    LPCTSTR m_pszHelpFilePath; // default based on module path
    LPCTSTR m_pszProfileName; // default based on app name
```

```
// Initialization Operations omitted

public:
    // Cursor wrappers
    // Icon wrappers
    // INI and profile wrappers

// Running Operations - to be done on a running application
    // Functions for dealing with document templates
    // Functions for dealing with files
    // Printer helpers
    // Command line parsing functions

// Overridables
    // hooks for your initialization code
    virtual BOOL InitApplication();

    // Functions to use when exiting
    // Advanced: MessageBox/Cursor overrides
    // Advanced: Help support

// Command Handlers
protected:
    // Command handlers for OnFileNew(), OnFileOpen(), and OnFilePrintSetup()
    // Context sensitive help functions

protected:
    // General attributes
    // Document/view, profile, and printer attributes and helpers

public: // public for implementation access
    CCommandLineInfo* m_pCmdInfo;

    // other attributes
    // Advanced OLE members

    void SetCurrentHandles();

    // helpers for standard commdlg dialogs
    // overrides for implementation
    virtual BOOL InitInstance();
    virtual int ExitInstance(); // return app exit code
    virtual int Run();
    virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing
    virtual LRESULT ProcessWndProcException(CException* e, const MSG* pMsg);

public:
    virtual ~CWinApp();
```

```

protected:
    //{{AFX_MSG(CWinApp)
    afx_msg void OnAppExit();
    afx_msg void OnUpdateRecentFileMenu(CCmdUI* pCmdUI);
    afx_msg BOOL OnOpenRecentFile(UINT nID);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};


```

CWinApp is chock full of member functions and variables. Here's a rundown of the most important ones:

- CWinApp maintains a set of the command line parameters passed into WinMain(). They are the current instance handle (*m_hInstance*), the previous instance handle (*m_hPrevInstance*), the command line parameters (*m_lpCmdLine*), and the show window flag (*m_nCmdShow*).
- CWinApp maintains a copy of the application name in *m_pszAppName*.
- CWinApp keeps pointers to the file name of the executable (*m_pszExeName*), the path to the application's help file (*m_pszHelpFilePath*), and the name of the application profiles (*m_pszProfileName*). We'll see how those are initialized shortly.
- CWinApp also uses a structure called CCommandLineInfo to retain command line parameters. Over the years, several standards for command line parameters have evolved. For example, when an OLE application is started as an object embedded in an OLE document, Windows starts that application with the command line argument "/Embedding". There are standard command line parameters for telling an application to perform such operations as open a new file, print an existing file, and run in Automation mode. CCommandLineInfo stores all the standard parameters in a single place. CCommandLineInfo is a small utility class, but it's handy if you ever need to get to the command line parameters. The definition of CCommandLineInfo is shown in Listing 2-4.

Listing 2-4. The defintion of CCommandLineInfo

```

class CCommandLineInfo : public CObject
{
public:
    // Sets default values
    CCommandLineInfo();

    virtual void ParseParam(const char* pszParam, BOOL bFlag, BOOL bLast);

    BOOL m_bShowSplash;
    BOOL m_bRunEmbedded;
};


```

```

BOOL m_bRunAutomated;
enum { FileNew, FileOpen, FilePrint, FilePrintTo, FileDDE } 
    m_nShellCommand;

// not valid for FileNew
CString m_strFileName;

// valid only for FilePrintTo
CString m_strPrinterName;
CString m_strDriverName;
CString m_strPortName;

~CCommandLineInfo();
};


```

- Every application needs to have a place where it can perform instance-specific initializations. In an MFC application, that place is CWinApp::InitInstance(). MFC calls InitInstance() before anything substantial happens in an application. For example, this is usually where the main window is shown.
- Conversely, most applications need a place to perform shutdown and to clean up code. In an MFC application, ExitInstance() serves this purpose.
- In MFC applications, the message pump is part of the CWinApp. Calling CWinApp::Run() starts a standard GetMessage()..DispatchMessage() loop. However, CWinApp's message loop has explicit support for performing background processing.
- One feature of an event-driven environment like Windows is that there may be moments in the life of an application when there are no messages in the queue. MFC takes advantage of this fact by including a function called OnIdle() in the CWinApp class. CWinApp::Run() calls OnIdle() whenever the message queue is empty.

That's a quick rundown of CWinApp. Now let's take a look at CWnd, the base class for all MFC windows.

CWnd: The Base Window Class

In addition to the CWinApp-derived object, an MFC Windows application contains an object representing the windows on the screen, derived from a class appropriately named CWnd. CWnd is way too big to show here. If you'd like to see the definition of CWnd in the flesh, just look in AFXWIN.H. CWnd provides two areas of functionality: (1) wrapping the regular Windows API functions (like Create() and ShowWindow()) and (2) providing higher-level MFC-related functionality, like default message handling.

Wrapping the Windows API

Wrapping the API functions is the easy part. CWnd maintains a member variable called `m_hWnd`, which represents a regular API-level window handle (you know, an `HWND`). For starters, CWnd encapsulates all the Windows API functions that take a window handle. This includes such functions as `ShowWindow()` and `MoveWindow()`. Basically, anytime you see an API function that takes a window handle (an `HWND`) as the first parameter, you can bet it's a member of CWnd. Anytime some client code calls a Windows API function in a CWnd-derived class, the CWnd version of the function uses the standard API function passing the object's window handle (`m_hWnd`). Nothing too exciting there. However, working with the Windows API at this level greatly reduces the surface area to which you have to pay attention. For example, here's the code to show a window using C and the SDK:

```
HWND hWnd
...
ShowWindow(hWnd, SW_SHOWNORMAL);
```

Doing the same thing in MFC looks like this:

```
CWnd* pWnd;
...
pWnd->ShowWindow(SW_SHOWNORMAL);
```

The MFC source code includes a file called AFXWIN2.INL (the INL extension stands for “inline”), which contains the code for CWnd’s default behavior. Most of CWnd’s Windows API wrappers are inlined. Some APIs are no longer inlined because they require special handling when the CWnd refers to an OLE control—they simply became too big to be inlined. In the case that the CWnd is not an OLE control, member functions are very simple calls into the Win32 API. Each CWnd member that maps to a Windows API function simply calls the API function using `m_hWnd`.

However, CWnd is more than just a wrapper for a bunch of API functions. CWnd retains other features, such as encapsulating the default window behavior for MFC-based programs.

Other CWnd Features

First, notice the derivation of CWnd:

```
CObject->CCmdTarget->CWnd
```

CWnd’s parent is CCmdTarget, which is in turn derived from CObject. By deriving from CObject, CWnd retains some compelling features, like dynamic run-time information and serialization. We cover these topics in Chapter 5. By deriving from

CCmdTarget, CWnd is able to hook into MFC's message-routing scheme, which is the focus later in this chapter.

In addition, CWnd defines MFC's default response to most window messages. Take another look at the switch statement for the C/SDK example in Listing 2-1. If the message is never handled, then MINIMAL's window procedure delegates to the default window procedure, DefWindowProc(). MFC deals with unwanted window messages the same way. Remember that MFC handles messages somewhat differently from a standard SDK application. MFC maps messages to handler functions. CWnd has members to handle most window messages. They come in the form of "On...()". For example, CWnd handles the WM_PAINT message in a function called OnPaint(). A quick examination of AFXWIN2.INL reveals that CWnd's message-handling functions simply delegate to CWnd::Default(). This results in a call to CWnd::DefWindowProc(), which is the end of the line for most messages.

Before moving on and digging deeper into the bowels of MFC, let's stop to examine an often misunderstood aspect of Windows development using MFC: the difference between window handles and the C++ classes that encapsulate them.

Turning Window Handles into Window Objects

Understanding the difference between native window handles (HWNDs) and the MFC objects representing windows is very important. At the highest level, MFC is set up to work with CWnd objects. However, native Windows code deals with window handles. For example, when Windows calls a window procedure, Windows passes a window handle as the first parameter. However, MFC's message dispatch mechanism works with CWnd-derived objects. In order for the message dispatching to work, MFC has to figure out which CWnd-derived object is associated with a particular handle. MFC uses a class called CHandleMap to relate CWnd-derived objects to window handles.

CHandleMap: The Undocumented Window Handle Map Class

The CHandleMap class maps window handles to MFC Windows objects. This means that given a handle, you can obtain the object associated with that handle. MFC needs a mechanism like this for precisely the conundrum just described: Windows uses handles and MFC uses objects. When Windows calls a callback function, it passes a window handle as a parameter. MFC needs to translate that into something it can deal with: a CWnd-derived class.

It's important for MFC objects to wrap native window handles whenever possible—that's the way MFC works. Because MFC frequently mixes native handles with MFC wrappers, the framework requires a uniform mapping between window handles and the C++ objects that wrap them.

The CHandleMap carries two members of type CMapPtrToPtr. They are called `m_permanentMap` and `m_temporaryMap`. CHandleMap uses the CMapPtrToPtr capabilities to maintain the relationship between window handles and their associated MFC objects. The permanent map, `m_permanentMap`, maintains the handle/object map for the life of a program. The temporary map, `m_temporaryMap`, exists for the duration of a message.

The permanent map stores those C++ objects that have been explicitly created by the developer. Whenever a CWnd-derived class is created, MFC inserts the mapping into the permanent dictionary. The mapping is removed whenever CWnd's `:OnNcDestroy()` is called.

Again, MFC deals with a CWnd-derived class instead of the actual HWND. Sometimes a handle passes through an MFC interface that doesn't exist in the permanent dictionary (that is, it wasn't explicitly created by the developer). In such a case, MFC allocates a temporary object to wrap the handle and stores that mapping in the temporary dictionary. These temporary wrapper objects are discarded from the temporary map at idle time.

A good example of this is CWnd::GetActiveWindow(). This is a static member function of CWnd that wraps the GetActiveWindow() API function. However, rather than return the window handle of the active window (as ::GetActiveWindow() does), CWnd::GetActiveWindow() returns a pointer to a CWnd object. Of course, the current active window may be in another application altogether, in which case CWnd::GetActiveWindow() has to create a temporary CWnd object to wrap the window handle. CWnd::GetActiveWindow() simply calls the global version of GetActiveWindow() to get a window handle. Then ::GetActiveWindow() uses CWnd::FromHandle() to create a CWnd object that wraps the active window handle. CWnd::FromHandle() looks for the window handle first in the permanent map and then in the temporary map. If CWnd::FromHandle() can find the handle in neither place, CWnd::FromHandle() allocates a CWnd object and attaches the handle to the object.

In addition to a mapping between CWnd-derived objects and HWNDs, there are four other mappings of MFC classes to native window handles. In all, MFC defines four of these maps, associated with each kind of handle-based class within MFC. The handle maps are located in the AFX_MODULE_THREAD_STATE structure, which we'll examine in more detail later. Here are all the handles to MFC object maps:

- `m_pmapHWND` maps between window handles and CWnd objects.
- `m_pmapHMENU` maps menu handles to CMenu objects.
- `m_pmapHDC` maps device context handles to CDC objects.
- `m_pmapHGDIOBJ` maps GDI object handles to CGDIObjects.
- `m_mapHIMAGELIST` maps image list handles to CImageList objects.

It's easy to retrieve one of these MFC classes given a native window handle. Each of the MFC classes participating in this handle-mapping scheme includes a FromHandle() function. The FromHandle() simply wraps CHandleMap::FromHandle(), which does the actual lookup to attach a native handle to one of the C++ objects. CWnd::FromHandle() returns an object of the appropriate type. For example, if you have a native window handle for which you'd like the CWnd object, just create a CWnd object and call FromHandle(). So, given a native handle type in Windows, you can always get to its corresponding C++ object using FromHandle().

Attaching and Detaching Window Handles

To complement the mapping mechanism between window handles and C++ objects, CWnd has a pair of functions for associating window handles and CWnd-derived objects. These two functions are Attach() and Detach().

CWnd::Attach() is fairly straightforward. Given a window handle, Attach() sets the value of CWnd::m_hWnd to the existing window handle and puts the association in MFC's permanent window handle map. CWnd::Detach() does the opposite. It removes the association between the window handle and the CWnd-derived object from the window handle map and sets CWnd::m_hWnd to NULL.

Using Attach() and Detach() is preferable to simply setting the member variable m_hWnd. Remember that the association between handles and C++ objects is managed by the system, so you should use MFC functions to associate window handles to C++ objects.

At this point, you know enough about CWinApp and CWnd to understand the rest of how an MFC application starts up and is initialized. Time to dig a little deeper.

Find WinMain() Now

CWinApp and CWnd can do little by themselves. The two objects work together in a symbiotic relationship that produces a working Windows application. At this point, you can see CWnd and CWinApp in the example program (Listing 2-2), but you have to take it on faith that all the other requirements are being fulfilled. You may be asking yourself, "Where exactly do the window classes get registered? When does the message loop get started? Where is the message handler?" Rest assured, all of that is happening inside the program (otherwise, it wouldn't run). Here's a deeper look at MFC and how it implements these basic elements of a Windows application.

In an MFC program, the game kicks off in the WinMain() function. Take another look at the minimal MFC program in Listing 2-2. The most obvious thing that's

missing from MINIMAL.CPP is the WinMain() function. Because the MFC program works, there must be a WinMain() in there somewhere. It just happens to be buried inside MFC.

You'll find MFC's WinMain() function in the file APPMODUL.CPP. Here's the WinMain() that MFC includes in your MFC application:

```
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPTSTR lpCmdLine, int nCmdShow)
{
    return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

As you can see from the code, WinMain() delegates processing to a function called AfxWinMain(). If you go look at the file WINMAIN.CPP, you'll see AfxWinMain(). It looks like a regular WinMain() function, and you can see it's actually doing WinMain() types of activities. Listing 2-5 shows AfxWinMain.

Listing 2-5. The AfxWinMain pseudocode, from WINMAIN.CPP

```
int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPTSTR lpCmdLine, int nCmdShow)
{
    ASSERT(hPrevInstance == NULL);

    int nReturnCode = -1;
    CWinApp* pApp = AfxGetApp();

    // AFX internal initialization
    if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
        goto InitFailure;

    // App global initializations (rare)
    ASSERT_VALID(pApp);
    if (!pApp->InitApplication())
        goto InitFailure;
    ASSERT_VALID(pApp);

    // Perform specific initializations
    if (!pApp->InitInstance())
    {
        if (pApp->m_pMainWnd != NULL)
        {
            TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
            pApp->m_pMainWnd->DestroyWindow();
        }
        nReturnCode = pApp->ExitInstance();
    }
}
```

```
    goto InitFailure;
}
ASSERT_VALID(pApp);

nReturnCode = pApp->Run();
ASSERT_VALID(pApp);

InitFailure:

AfxWinTerm();
return nReturnCode;
}
```

Compare this WinMain() with other WinMain() functions you might find elsewhere (such as in the SDK samples). There's no mystery here. The parameters are all there and the general functionality is the same. The one thing you may notice is the “_t” in front of WinMain(). The “_t” is used for Unicode support. Let's take a look at some important structures used by the framework before stepping through WinMain().

MFC State Information

It's often useful for an application to maintain certain information throughout the life of the program. For example, most standard C/SDK programs save the instance handle in a global variable. There are often times when an application needs to use the instance handle to retrieve resources bound to the EXE file, and this is a standard way to save the handle. MFC does these same things and more by maintaining various structures throughout the life of the program.

AFX_MODULE_STATE: Undocumented State Information

In contrast to most C/SDK programs, MFC applications keep a great deal more state information than just an instance handle. As an application framework, MFC has to be as generic as possible—it may potentially be required to cover much more ground than a single specific application written in C and the SDK. Like most other Windows applications, MFC applications have to keep track of various items, including main window handles, resources, and module handles. Think of everything MFC provides, including such items as memory allocation tracking, ODBC support, OLE support, and exception handling. To support these features, MFC maintains much more information than a regular Windows application might. To this end, MFC defines a structure called AFX_MODULE_STATE in the AFXSTAT_.H file, which is shown in Listing 2-6.

Listing 2-6. The AFX_MODULE_STATE pseudocode, from AFXSTAT.H

```

class AFX_MODULE_STATE : public CNoTrackObject
{
public:
#ifndef _AFXDLL
    AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion);
    AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion,
                     BOOL bSystem);
#else
    AFX_MODULE_STATE(BOOL bDLL);
#endif

    CWinApp* m_pCurrentWinApp;
    HINSTANCE m_hCurrentInstanceHandle;
    HINSTANCE m_hCurrentResourceHandle;
    LPCTSTR m_lpszCurrentAppName;
    BYTE m_bDLL; // TRUE if module is a DLL, FALSE if it is an EXE
    BYTE m_bSystem; // TRUE if module is a "system" module, FALSE if not
    BYTE m_bReserved[2]; // padding

    short m_fRegisteredClasses;// flags for registered window classes

    // runtime class data
#ifndef _AFXDLL
    CRuntimeClass* m_pClassInit;
#endif
    CTypedSimpleList<CRuntimeClass*> m_classList;

    // OLE object factories
#ifndef _AFX_NO_OLE_SUPPORT
#ifndef _AFXDLL
    COleObjectFactory* m_pFactoryInit;
#endif
    CTypedSimpleList<COleObjectFactory*> m_factoryList;
#endif

    // number of locked OLE objects
    long m_nObjectCount;
    BOOL m_bUserCtrl;

    // AfxRegisterClass and AfxRegisterWndClass data
    TCHAR m_szUnregisterList[4096];
#endif _AFXDLL
    WNDPROC m_pfnAfxWndProc;
    DWORD m_dwVersion; // version that module linked against
#endif

    // define process local/thread local portions of module state
    PROCESS_LOCAL(AFX_MODULE_PROCESS_STATE, m_process)
}

```

```
THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)  
};
```

AFX_MODULE_STATE contains core information about the module—that is, information required by all MFC modules regardless of type (whether EXE or DLL). Collected within this structure are the module instance handle, the instance of the module from which to pull resources, a pointer to the module's CWinApp-derived class, the name of the application, a pointer to the first node in the application's list of run-time class information structures, and a pointer to an exception handler. The exception handler is only for compilers without exception handling. In general, it is not used with modern compilers (such as VC++ 2.x and above).

Here's a rundown of the most important AFX_MODULE_STATE members:

- m_pCurrentWinApp—A pointer to a CWinApp.
- m_hCurrentInstanceHandle—The instance handle of this module.
- m_hCurrentResourceHandle—Represents the instance handle holding the module's resources.
- m_lpszCurrentAppName—A pointer to the application's name.
- m_bDLL—Indicates whether the module is a dynamic link library or an executable.
- m_classList—Points to the first run-time class in the application's list of CRuntimeClass structures.
- m_factoryList—Points to the first run-time class in the application's list of COleObjectFactory structures.
- m_nObjectCount—A reference count for OLE servers indicating whether the server has outstanding COM objects.
- m_bUserCtrl—A flag indicating whether or not an OLE server is in use by the user.
- m_szUnregisterList[4096]—Maintains a list of registered window classes so that MFC can unregister them upon termination.
- m_pfnAfxWndProc—Points to MFC's standard window procedure.
- m_fRegisteredClasses—Indicates which MFC window classes have already been registered.

Developer Tips

Take a closer look at the AFX_MODULE_STATE again. Notice that the structure contains two instance handles: an application instance and a resource instance. This is useful if you want to internationalize your application. Whenever you employ a user-interface resource (such as a dialog box template or a string table), MFC uses the module's resource instance

(AFX_MODULE_STATE::m_hCurrentResourceHandle) to retrieve the resource. Normally, this resource handle is set to the application's EXE file. However, there's no reason why you can't use a separate instance handle (perhaps that of a DLL) to retrieve resources customized for a certain locality.

Notice that AFX_MODULE_STATE also contains a pointer to the first node in a list of CRuntimeClass structures. This list is initialized as soon as the module starts and is built up as new classes are derived from CObject. If you ever need to enumerate the run-time class information for your application, this is the place to start. Of course, the caveat is to be careful about this stuff (the structures in AFXSTAT_.H). They are all subject to change in the future. Though the state and status structures remain useful for debugging purposes, it's better not to write code depending on them.

As you can see, MFC holds a great deal more information than a mere instance handle. Now that you know that these structures exist, you know where to look if you think you need access to some bit of state information about your application. Particularly useful are the resource instance handles and the run-time class information list in AFX_MODULE_STATE. Many of these members are accessible via documented functions like AfxGetInstanceHandle(). For now, it's just important to be aware of the data kept in these structures. We'll revisit most of these members later.

Of course, the caveat remains here: These are *undocumented* structures. They are subject to change. For example, version 1.5's CRuntimeClass structure contained a static pointer to the application's run-time class list. As you can see in Listing 2-6, the pointer has been moved to the AFX_MODULE_STATE structure. As you work with these structures, just beware—especially if you assign values to the fields.

Version Note

MFC version 3.0 split this information into separate structures: AFX_CORE_STATE, AFX_WIN_STATE, and AFX_OLE_STATE. Version 3.0 also has a structure called AFX_MODULE_STATE, which simply nests the three state structures.

Now that you have an idea of the state information maintained by MFC, let's continue comparing the MFC approach to the C/SDK approach.

Back to WinMain()

Now that you understand the structures MFC uses to maintain module state information, AfxWinMain() should make a little more sense. Let's get back to watching a minimal MFC program run. The first thing AfxWinMain() does is declare a pointer variable of type CWinApp and assign the value of AfxGetApp() to it. The name AfxGetApp() is self-describing: it gets the single application object associated with

the program (a CWinApp-derived object is required by every MFC program). However, if WinMain() is the entry point to the Windows application, and the application object isn't constructed within WinMain(), how can the framework retrieve the application object?

The answer lies in the way C++ handles constructing global objects. C++ programs construct their global objects before anything else is done—even before main() (or even WinMain()) is called. As long as you include a global CWinApp-derived object in your program, you can rest assured that it will be constructed by the time WinMain() gets around to executing.

Constructing CWinApp

CWinApp's constructor code is in APPCORE.CPP. The main job of CWinApp's constructor is to initialize CWinApp's member variables. CWinApp's constructor takes a single parameter: the name of the program. CWinApp sets its m_pszAppName variable to the value passed in. This parameter defaults to NULL—you can choose to provide your own application name if you like. Then CWinApp's constructor initializes the module's thread state and module state structures (hold tight: we'll cover the AFX_THREAD_STATE structure in Chapter 10). CWinApp's constructor initializes the AFX_MODULE_STATE's m_pCurrentWinApp to the CWinApp being constructed. All of CWinApp's other members—handles, pointers, and strings—are set to NULL (that is, the instance handle, the pointer to the main window, the name of the application, and so on).

Initializing the Framework: AfxWinInit()

Next, AfxWinMain() calls AfxWinInit() to initialize the framework. AfxWinInit() takes the same four parameters as WinMain(): the current instance handle, the previous instance handle, the command line parameters, and the show command.

AfxWinInit() sets the error mode for the application using SetErrorMode(). This designates what will cause the application to fail. SetErrorMode() takes a bitmap flag. MFC sets the error mode using the SEM_FAILCRITICALERRORS and SEM_NOOPENFILEERRORBOX flags. By using SEM_FAILCRITICALERRORS, MFC tells Windows not to display the critical-error-handler message box for critical errors, but rather to send the error to the calling process. The SEM_NOOPENFILEERRORBOX flag causes Windows to not display a message box when it fails to find a file. Instead, the error is returned to the calling process.

AfxWinInit() then calls AfxGetModuleState() to get the module's AFX_MODULE_STATE structure. AfxWinInit() stores the module instance handle and the resource handles in AFX_MODULE_STATE::m_hCurrentInstanceHandle and AFX_MODULE_STATE::m_hCurrentResourceHandle. At this point both the current instance handle and the resource handle point to the module's instance handle.

Whenever a Windows application starts, Windows passes four parameters to the application: (1) the handle to the module instance, (2) the handle to the previous instance of the module, (3) any command line parameters, and (4) the show window flag. CWinApp keeps copies of these parameters as member variables. AfxWinInit() sets CWinApp::m_hInstance, CWinApp::m_hPrevInstance, CWinApp::lpCmdLine, and CWinApp::nCmdShow. Then AfxWinApp() calls the application object's SetCurrentHandles() function and sets the AFX_MODULE_STATE handles again. This redundancy is historical. The module state structure didn't used to exist—it was introduced to handle OLE controls. It is possible that in the future the ones in CWinApp will go away, depending on backward-compatibility issues.

Next, CWinApp::SetCurrentHandles() initializes the application name and path variables within CWinApp: First, SetCurrentHandles() uses GetModuleFileName() to retrieve the module file name. GetModuleFileName() is a Win32 API function that returns the fully qualified name of a running instance of a module. SetCurrentHandles() initializes CWinApp::m_pszExeName to the name of the executable file (without the EXE extension). For example, if you wrote an MFC application called "FILEVIEW.EXE", CWinApp::m_pszExeName would have the value "FILEVIEW". SetCurrentHandles() then initializes the m_pszAppName to the title of the application. If you use AppWizard to create your project, MFC uses a specific string from your project's resource file bearing the ID AFX_IDS_APP_TITLE. If this resource is unavailable, m_pszAppName is initialized to the same value as m_pszExeName. SetCurrentHandles() also sets the application's AFX_MODULE_STATE::m_lpszCurrentAppName to the same value as m_pszAppName.

SetCurrentHandles() also fills CWinApp's help file and profile strings: m_pszHelpFilePath and m_pszProfileName. SetCurrentHandles() initializes m_pszHelpFilePath to the same value returned by GetModuleFileName() but uses HLP as the extension. SetCurrentHandles() initializes m_pszProfileName to the same name as the application, but with an INI extension.

If AfxWinInit() has accomplished everything listed so far without a hitch, the application and the framework are both initialized properly.

Now that the handles and file names are all initialized correctly, MFC continues with the rest of the application. AfxWinMain() calls the application's InitApplication() function.

InitApplication()

InitApplication()'s job is to perform any initializations that pertain to the entire application. CWinApp's version of InitApplication initializes the application's document manager. (You'll see more of this in Chapter 7—all about the document/view architecture.) Note that when Windows moved to 32 bits, InitApplication() became somewhat obsolete. All initialization should now take place in InitInstance(). InitApplication()

made sense only in 16-bit Windows, where there was a difference between the first instance of an application and subsequent ones (which you could determine by examining the hPrevInstance). Win32 doesn't distinguish between first and subsequent instances.

AfxWinMain() then initializes the specific instance of the application using InitInstance().

InitInstance()

Whenever a program begins, it's often necessary to perform initializations for a certain instance of the program. CWnd::InitInstance() serves that purpose. CWinApp's default implementation of InitInstance() does nothing. It just returns TRUE. However, InitInstance() is also virtual, so you can safely override it. Activities that take place inside InitInstance() include such tasks as setting all the documents for an application and showing the main window. Because the default version of InitInstance() doesn't do anything, it's up to you to make sure a window appears on the screen.

Priming the Message Pump: CWinApp::Run()

The last thing WinMain() does before leaving is call the CWinApp-derived object's Run() function. Run() starts the ball rolling with the message loop. We'll see in a moment that the Run() function does a little more than just a generic GetMessage()..DispatchMessage() loop.

Some Other Hidden Cool Stuff

Before we examine the message pump, let's consider two more pieces to the puzzle. One is obvious and the other is not so obvious. The obvious issue is window classes: all Windows applications have to register at least one window class. The less obvious issue is Windows hooks and how the MFC window procedure is wired in. Let's start by looking at the MFC window classes. (That's *window* classes, which is different from *C++* classes.)

Registering Window Classes

Before a Windows application can display a window, the application has to register at least one window class with the operating system. An MFC application is just like any other Windows application, so it needs to register at least one window class as well. A window class defines very basic aspects of a window, such as its appearance

(via some flags) and its behavior (via a callback function). MFC actually registers four standard window classes. MFC registers a window class for (1) regular child windows, (2) a control bar window, (3) an MDI frame window, and (4) a window for an SDI or MDI child window. The names are decorated with the MFC version number. In addition, MFC uses information about whether the app is statically linked and information about whether the app is a debug or release build of MFC to decorate the class names. These are the four decoration window class names supplied for nonstatic debug builds of an application:

- AfxFrameOrView40d
- AfxControlBar40d
- AfxWnd40d
- AfxMDIFrame40d

MFC also has a new window class for the common controls, which we'll cover in Chapter 6.

MFC registers its window classes in an interesting way. It defines a macro called `AfxDeferRegisterClass()` in `AFXIMPL.H`:

```
#define AfxDeferRegisterClass(fClass) \
    ((afxRegisteredClasses & fClass) ? TRUE : \
     AfxEndDeferRegisterClass(fClass))
```

`AfxDeferRegisterClass` works by first finding out whether a specific window class has already been registered. The global variable `afxRegisteredClasses` maintains a bitmap representing MFC's window classes. The following values represent MFC's window classes:

<code>AFX_WND_REG</code>	<code>(0x0001)</code>
<code>AFX_WNDCONTROLBAR_REG</code>	<code>(0x0002)</code>
<code>AFX_WNDMDIFRAME_REG</code>	<code>(0x0004)</code>
<code>AFX_WNDFRAMEORVIEW_REG</code>	<code>(0x0008)</code>

MFC uses the global `afxRegisteredClasses` variable to optimize the window registration. This is important because MFC attempts to register the window classes in many different places. `AfxDeferRegisterClass()` first checks against the indicators for previously registered classes to determine if it should take the time to register the window class. If the class is already registered, then `AfxDeferRegisterClass()` simply returns `TRUE`. Otherwise, `AfxDeferRegisterClass()` attempts to register the window class represented by the just-mentioned bitmap.

Version Note

MFC version 3.0 registered all its window classes during AfxWinInit().

AfxEndDeferRegisterClass()

AfxDeferRegisterClass() zeroes out a WNDCLASS structure using memset so that all fields except those being set explicitly are NULL or zero. AfxDeferRegisterClass() initializes WNDCLASS::lpfnWndProc to DefWindowProc(), the same DefWindowProc() you stick at the end of a message loop to deal with messages that are irrelevant to your application. Even though it seems as if that would prevent your application from receiving messages, MFC has a way of directing messages to your application. You'll find out how in Chapter 3: Message Handling in MFC. The window class's hInstance handle is the current instance handle. AfxDeferRegisterClass() sets the window class's cursor to the regular arrow cursor. These values are common among all MFC window classes.

Once the class structure has been zeroed out and the common fields initialized, AfxEndDeferRegisterClass() starts filling it, depending on the window class being registered. If AfxEndDeferRegisterClass() is trying to register a regular, plain-vanilla window, the window class is called "AfxWndXXXX" (decorating the class name appropriately). It uses a minimal set of class styles: CS_DBCLKS, CS_HREDRAW, and CS_VREDRAW. All other fields are NULL or zero. This window class is used for all CWnd objects created using CWnd::Create().

If the caller is trying to register a toolbar, AfxEndDeferRegisterClass registers the "AfxControlBarXXXX" class. MFC turns the class style bits off—the style is set to zero. That is, the CS_HREDRAW and CS_VREDRAW are turned off because control bars (toolbars and status bars) don't need to redraw the complete window when a WM_SIZE message is received. This also reduces flashing when the window is being redrawn. In addition, control bars don't handle double-clicks. Finally, the background brush of this window class is set to the button face color.

If the caller is registering an MDI frame window class, it receives the name "AfxMDIFrameXXXX". AfxEndDeferRegisterClass() turns on the double-clicks flag (CS_DBCLKS) and sets the background brush to NULL. The window class then is registered with the default MDI frame icon provided by AppWizard, which has the identifier AFX_IDI_MDI_FRAME. Notice that if you want to change the icon for an MDI frame window, you just need to change the AFX_IDI_STD_MDIFRAME icon.

The final window class for SDI frame windows or MDI child windows or views is registered with the name "AfxFrameOrViewXXXX". It has a style of CS_DBCLKS, CS_HREDRAW, and CS_VREDRAW. The background brush is the default color for a window. The window class is registered with the default SDI frame icon, called AFX_IDI_STD_FRAME. Again, as with the MDI frame window class, if you want

to change the frame icon, you just need to change the AFX_IDI_STD_FRAME icon in your application.

So, when are these window classes registered? Earlier versions of MFC registered these classes as part of the startup code. The following list shows which functions register window classes and which classes they register:

- CDialogBar::Create()—Registers the AfxControlBar class.
- CDockBar::Create()—Registers the AfxControlBar class.
- COleResizeBar::Create()—Registers the AfxControlBar class.
- CView::PreCreateWindow()—Registers the AfxFrameOrView class.
- CWnd::Create()—Registers the AfxWnd class.
- CFrameWnd::PreCreateWindow()—Registers the AfxFrameOrView class.
- CFrameWnd::LoadFrame()—Registers the AfxFrameOrView class.
- CMDIFrameWindow::PreCreateWindow()—Registers the AfxMDIFrame class.
- CSplitterWnd::CreateCommon()—Registers the AfxMDIFrame class.

Microsoft decided to register the window classes during window creation rather than all at once during the beginning of an application because Wnd class registration does take time. MFC avoids registering window classes that aren't necessary. In some cases only a few (or even none) of those classes are used (for example, consider an OLE control or a simple DLL with no user interface). This optimizes startup time for these scenarios.

MFC's Windows Hooks

AfxWinInit() performs an interesting step as part of initializing the application: it installs two hooks. The first hook is a message filter hook (as signified by the WH_MSGFILTER flag in the call to SetWindowsHookEx()). A message filter hook monitors messages generated as a result of input events in a dialog box, message box, menu, or scroll bar. MFC installs the function _AfxMsgFilterHook() as the message hook. The reason for this hook will become evident when we go through the command and message-routing architecture in the next chapter.

The second hook is to support computer-based training applications. MFC sets this hook by calling SetWindowsHookEx() using the WH_CBT flag.

MFC does not specifically support computer-based training. MFC installs this hook just so that MFC can hook in as soon as a window is created. MFC can't wait until CreateWindowEx() returns, because by then a number of messages have been sent. The CWnd object has to be hooked up and subclassed before any messages

are delivered to the window. The WH_CBT hook allows this (although that wasn't its intended purpose originally).

Clearly, MFC performs the steps necessary to qualify an application to run under Windows. It provides a WinMain() entry point, it provides a message handler, and it registers some window classes. The only thing left for an MFC application to do is to start a message pump. That happens in CWinApp::Run().

MFC's Message Pump: CWinApp::Run()

One of the main features offered by MFC version 3.0 (and greater) is safe user-interface threads. (In the past, you couldn't use threads within user-interface code.) (See Chapter 10 for more details.) To accomplish thread safety, Microsoft defines two different kinds of threads: (1) standard, Win32 threads (a.k.a. "worker" threads) and (2) MFC interface threads. Both types of threads are encapsulated by the CWinThread class. The main difference between worker threads and interface threads is that the worker threads are fully preemptive. That is, they are real Win32 threads created by the CreateThread() API. On the other hand, interface threads are simply message pumps (that is, the regular Windows GetMessage()..DispatchMessage() loop). Because CWinApp is derived from CWinThread, CWinApp inherits a message pump from CWinThread. You'll find the message pump implemented within CWinApp's Run() function.

Recall that the last step within WinMain() is a call to the application's Run() function. At this point, the CWinApp-derived class simply defers to the CWinThread's Run() function to start the message pump. If you want to find out exactly how CWinThread::Run() works, jump ahead to Chapter 10. Right now, all you need to know is that messages are processed the same way as in any other Windows program: using GetMessage(), TranslateMessage(), and DispatchMessage(). However, MFC handles message routing a bit differently. Instead of a huge switch statement to filter out a window's messages, MFC uses an approach called message maps, which we'll see in the next chapter.

For now, let's examine another basic concept in Windows programming: the Graphics Device Interface, or GDI.

MFC's GDI Support

One of the more compelling reasons for developers to write applications for Windows is its support for graphics. Windows has always had a fairly rich supply of graphics tools, and with the addition of bezier and path functions in Win32, the support has gotten much better.

One of the original design goals of Windows was to maintain device independence. The idea was to hide graphic rendering behind a single API. Doing so allows developers to use a single set of graphics functions regardless of the device performing the actual rendering. For example, the developer should be able to draw on a screen, a plotter, or a printer using the same functions. This API is the Graphics Device Interface.

The basic idea behind the GDI is that the operating system supports a single set of functions and data structures for drawing on various devices. Using the GDI at the highest level is the application itself. The application uses GDI functions to draw on some device (usually the screen or printer). However, rather than send the drawing commands directly to the device, Windows sends the commands to a device driver—a piece of software sitting between Windows and the physical device. When the device driver receives the drawing commands, the device driver does whatever is necessary to perform the rendering on the physical device. Each device manufacturer is supposed to provide the Windows-compatible driver for its own hardware.

This concept works very well. It's far and away better than the way this was done in DOS, where programs wrote directly to the physical device. However, programming the GDI isn't the simplest undertaking in the world. In fact, it can be downright cumbersome and tedious. MFC seeks to simplify GDI programming by helping to mask some of the details and inconsistencies of the GDI.

Device Contexts

The heart of the GDI is the device context. The idea behind producing output under Windows is that you write to a standard API (the Graphics Device Interface functions), and Windows handles the rest. The Windows graphics API centers around a structure called the device context. When generating output under Windows, you direct output to a device context. The device context might represent a screen (which is the most common case), or it might be some other device, such as a printer or a plotter. From a programming point of view, you use the same functions to generate output to a screen device context as you do to generate output to a printer device context.

Software Patterns: The Windows GDI and the Facade Pattern

One of the more exciting ideas to emerge in the area of software development is the notion of software patterns, which is explored thoroughly in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995). This work has two parts: (1) a well-written description of just what software patterns are and how they're useful and (2) a catalog of 23 software design patterns.

(continued)

In short, software patterns are formalized descriptions of proven designs. Software designs can be thought of as “first principles.” For example, most experienced software designers tend to hide their code behind a solid, well-defined interface. That way, they can change the implementation or write similar components that “plug right in.” This is the idea behind the Windows GDI. With the GDI, you use one set of functions to write to any number of media: the screen, the printer, a metafile, and so on. Windows handles the rest by delegating to a device driver, which in turn talks to the actual device. In fact, the idea behind the Windows GDI is basically the “facade” pattern described in *Design Patterns*. Be sure to check it out.

Device contexts define a set of graphic objects and their associated attributes (for example, pens, brushes, and bitmaps) and the graphic modes that affect output (like the background mode and the mapping mode).

Of course, MFC has several classes for representing device contexts. MFC defines several different kinds of device contexts: one for painting (CPaintDC), one for representing the entire screen area of a window (CWindowDC), one for representing the client area of a window (CCClientDC), and one for specifically metafiles (CMetaFileDC).

MFC’s support for device contexts is similar to its support for regular window handles. That is, the MFC device context classes (CDC, CCClientDC, CPaintDC, CWindowDC, and CMetaFileDC) all wrap a Windows device context handle (an HDC). As with the window handle-related API functions, whenever you see an API function whose first parameter is an HDC, that API function is probably a member of one of the device context classes.

In addition, MFC maintains a mapping between HDCs and their corresponding CDC-derived objects. This map is a member of AFX_MODULE_THREAD_STATE. CDC has a member function called FromHandle() that retrieves the C++ object matching a specific HDC.

Inside the CDC Class

As you might suspect, the CDC class maintains a member variable of type HDC, which represents a real device context handle. However, CDC also contains another HDC, called m_hAttribDC. This device context is usually equal to m_hDC. CDC functions that request information from the device context usually use m_hAttribDC.

CDC also contains a BOOL member variable indicating whether or not the output is going to a printer.

The CDC constructor is very simple: it just initializes the member variables (m_hDC, m_hAttribDC, and m_bPrinting) to NULL (or FALSE for m_bPrinting). Attach() sets CDC::m_hDC to a specific HDC and adds the HDC/C++ object association to the map. CDC::Detach() removes the association from the map.

In addition to wrapping device context-related API functions, CDC contains some non-HDC-related functions. For example, the Windows API functions for

printing are member functions of CDC. This includes such functions as StartDoc(), EndDoc(), and Escape(). You'll see more of these printing functions in Chapter 8, on document/view architecture.

Finally, CDC's destructor calls DeleteDC(). Deleting the DC when you're done using it is critical because there is a limited supply of device contexts. Using the MFC device context classes takes care of deleting the device context when done.

CDC Derivations

MFC derives specific device contexts from CDC: CPaintDC, CWindowDC, CClientDC, and CMetaFileDC.

- **CPaintDC**—This class is used whenever painting occurs. It is the same device context retrieved by calling BeginPaint(). CPaintDC's constructor takes a pointer to a CWnd and creates a device context by calling BeginPaint() and attaching it to the device context map. CPaintDC's destructor calls EndPaint() and detaches the device context handle.
- **CWindowDC**—This structure represents the entire screen area of a window (both the client area and the frame). CWindowDC's constructor takes a pointer to a CWnd and uses GetWindowDC() to create a device context for the entire window. CWindowDC's destructor calls ReleaseDC().
- **CClientDC**—This device context represents the client area of a window. CClientDC's constructor also takes a pointer to a CWnd. CClientDC uses GetClientDC() to create a device context for the client area of the window. CClientDC's destructor calls ReleaseDC().
- **CMetaFileDC**—Windows metafiles contain a sequence of GDI commands that can be replayed to create an image. MFC defines a class—CMetaFileDC—that wraps the metafile device context. Creating a metafile DC requires two steps: First, construct the CMetaFileDC (by allocating one or declaring one on the stack). Then call CMetaFile::Create() to create the metafile and initialize CMetaFile's member variables. CMetaFileDC's destructor calls DeleteMetaFile().

The device context is only half the story. To do anything with graphics, you need to be able to specify what kinds of tools you're using to do the drawing. Those tools are represented by MFC's GDI object classes.

Graphic Objects

A device context maintains all the necessary information for drawing in Windows, including such information as the mapping mode. The other piece of the GDI puzzle is the graphic object used by a device context. The device context maintains information

about the tools being used to render images on the device. These drawing tools include fonts, pens, and brushes.

MFC's support for GDI objects consists of a base class called CGDIOBJECT and several classes representing the drawing tools. These include CPen, CBrush, CFont, CBitmap, CPalette, and CRgn.

CGDIOBJECT

This class represents the base class for all GDI objects. MFC's support for GDI objects is similar to that for window handles. GDI objects are defined by handles, and MFC maintains a mapping of GDI objects and the C++ objects that represent them. CGDIOBJECT has the member functions for managing and using these maps. CGDIOBJECT contains a FromHandle() function, as well as Attach() and Detach() functions.

CGDIOBJECT contains a member variable called m_hObject, which represents the handle to the GDI object. This handle could belong to a brush, a pen, a font, a bitmap, a palette, or a region.

As with window handles and device context handles, MFC maintains an association between GDI object handles and the C++ objects that wrap them. You'll find that CGDIOBJECT also has member functions for managing the association and retrieval of the handles. Fortunately, the functions carry the same names. That is, if you want to get the GDI object wrapping a specific GDI handle, you use FromHandle(). The GDI object/handle map is maintained in the AFX_MODULE_STATE structure described earlier.

How many times have you forgotten to delete a GDI object when you were done with it? Leaking resources is evil. Even now, when there's a much larger resource pool for your GDI objects, you should clean up after yourself. That includes deleting GDI objects when you're done with them. Fortunately, CGDIOBJECT's destructor calls DeleteObject().

MFC derives six GDI objects from CGDIOBJECT: CPen, CBrush, CFont, CBitmap, CPalette, and CRgn.

- CPen—Wraps the Windows pen object (an HPEN) and includes the API functions for creating pens as member functions.
- CBrush—Wraps the Windows brush object (an HBRUSH) and the API functions for creating brushes.
- CFont—Wraps the Windows font object (an HFONT) and the API functions for creating and managing fonts.
- CBitmap—Wraps the Windows bitmap object (an HBITMAP) and the functions for creating, loading, and managing bitmaps.
- CPalette—Wraps the Windows palette object (an HPALETTE) and the functions for creating and using palettes.

- CRgn—Wraps the Windows region object (an HRGN) and the functions for using regions.

Using these GDI objects is simply a matter of selecting them into a live device context and using them. For example, if you'd like to draw blue lines on the client area of your window, you could use OnPaint() as follows:

```
CMyWnd::OnPaint() {  
    CPaintDC paintDC(this);  
    CPen* pOldPen;  
    CPen bluePen(PS_SOLID, 25, RGB(0, 0, 255));  
  
    pOldPen = paintDC.SelectObject(&bluePen);  
    paintDC.MoveTo(1, 1);  
    paintDC.LineTo(100, 100);  
    paintDC.SelectObject(pOldPen);  
}
```

Conclusion

So there you have it—basic Windows application support within MFC. All Windows programs have to perform many of the same operations to run correctly. In addition, these steps are often the same regardless of the application being developed. MFC handles all that boilerplate code: what took 80 lines of C and SDK code can be done in 20 lines of C++ and MFC code. MFC handles the two basic parts of a Windows application within its CWinApp and CWnd classes.

However, MFC does more than just whisk away the necessary boilerplate code. MFC is a full-fledged application framework, full of goodies to make your life as a software developer easier. The next chapter covers MFC's message-mapping architecture in detail. In the following chapters, we'll examine MFC's utility classes (Chapter 4), the CObject class (Chapter 5), MFC's support for dialogs and controls (Chapter 6), MFC's document/view architecture (Chapters 7 and 8), and MFC's user interface support.

Message Handling in MFC

So far, we've covered the first major part of a Windows program: the main application itself. That's the part of the program that performs various initializations, shows the main window, and starts the message pump going. There's still the second part to cover: the actual message handling.

It's easy to see how messages are handled in a regular C/SDK program—just look for the big switch statement inside a window procedure. However, you won't find a big switch statement or a regular window procedure in an MFC program. So how are messages handled? When the application is initialized, each window class uses the same callback function—`DefWindowProc()`. How can there be only one callback function for *all* the windows in your application? And besides, `DefWindowProc()` doesn't do much anyway—its main use is to handle those orphaned messages that no one else wants. Why is that the procedure registered with the window MFC classes? Answers to these questions are coming up next as we dig deeper into MFC's message handling.

CCmdTarget and Message Maps

There are basically two components to MFC's message-handling architecture: (1) the `CCmdTarget` class and (2) message maps. These two pieces work together to accommodate the same message-handling capabilities as a regular C/SDK application. And once you understand message maps, you'll already have a grasp of the way MFC implements the Component Object Model, automation, and OLE control events, because they all use similar technology.

Message maps have been a feature of the Microsoft Foundation Classes ever since the first version came out with Microsoft C version 7. By connecting window

messages to the code that handles them, message maps replace the unruly switch statements that accompany Windows applications developed using C and the SDK.

Understanding the mechanics of MFC message maps is important for several reasons:

1. You can satisfy your own curiosity about MFC's internals.
2. You'll know why an MFC program is laid out the way it is (for example, you'll know why you should put the `DECLARE_MESSAGE_MAP` in an MFC class's header and the `BEGIN_MESSAGE_MAP`/`END_MESSAGE_MAP` macros in the call's implementation).
3. You'll better understand the message map syntax.
4. You'll have a better overall feeling for the gestalt of MFC.

Here we dissect MFC message maps so that you will understand the following:

1. How message maps replace the standard switch statement reminiscent of a C/SDK program.
2. What structures make up a message map.
3. Exactly how message maps work.
4. How messages flow through the application framework.

To fully understand message maps and the rationale for their development, it's important to know about window messages themselves and how Windows message handling has evolved. We'll start by examining window messages in their native form and how they are handled in a regular C/SDK application. Next, we see why MFC uses message maps instead of virtual functions for handling window messages. Finally, we break apart the components of a message map and then trace a command message (that is, one generated from a menu item or other user-interface object) and a regular window message through the framework to see exactly how message maps work.

Window Messages

Windows is an event-driven operating environment: It sits on the computer watching the hardware for events. Whenever something interesting happens, for example, when the mouse moves, Windows detects the event and passes that information on to the applications it is hosting. The information about that event is called a message. Specifically, in this case it's the `WM_MOUSEMOVE` message. If the message is

important to a specific window on the screen, the window handles the message. For instance, if Windows detects the selection of a certain menu option in a specific window on the screen, Windows passes that information to the window's message handler. The window's message handler contains a case that handles the menu option.

All window messages have the same basic form. That is, a window message has three components: (1) an unsigned integer containing the actual message, (2) a WPARAM—a word-size parameter (WPARAM is 32 bits in Win32), and (3) an LPARAM—a four-byte parameter. Microsoft defined window messages in such a general manner because Windows has no knowledge about specific cases for each message. This means that each message must exhibit polymorphic behavior. In other words, a window message has a single interface that has many different behaviors.

Of the three components of a window message, the unsigned integer always refers to the actual message. However, the WPARAM and the LPARAM can mean a variety of different things. In fact, the LPARAM component of a window message often contains additional data or points to a data structure required to handle the message.

Take a typical message like WM_COMMAND, for instance. WM_COMMAND is sent to a window whenever (1) a menu item is selected, (2) a control sends a notification code to its parent window, or (3) an accelerator keystroke is translated. The low word of the WPARAM parameter represents a number identifying the control. The high word of the WPARAM is the notification code, indicating things like whether the user double-clicked on the control. The LPARAM represents the window handle of the control sending the message.

This is just the information included with the WM_COMMAND message. Other messages may contain different information in their WPARAM and LPARAM parameters. Obviously, it can get rather difficult trying to keep track of the messages and what the parameters mean.

Windows starts generating messages the nanosecond it gets rolling. Naturally, programs need to have a way of filtering out messages so that they can respond appropriately. In the old SDK days, this was done using a window procedure with a switch statement to interpret the messages.

Message Handling Using C and the SDK

At the heart of any Windows program (even MFC-based programs) is a message pump. The message pump is a tight loop that picks up messages and drops them off with the appropriate window message handler. Here's a typical message pump:

```

while (GetMessage(&msg, NULL, NULL, NULL)) {
    TranslateMessage(&msg); /* Translates virtual key codes      */
    DispatchMessage(&msg); /* Dispatches message to window     */
}
return (msg.wParam);      /* Returns the value from PostQuitMessage */

```

When a C/SDK program begins, it immediately registers at least one window class using the RegisterClass() API function and a WNDCLASS data structure. In addition to such information as the icon and background brush to use for a window, the WNDCLASS data structure includes a pointer to a callback function for handling the window's messages. Whenever there is a message ready for a window of that class, Windows calls the function specified in the lpfnWndProc field of the WNDCLASS structure.

In a classic C/SDK program, the message handler usually turns out to be a huge switch statement that filters out any message that the window might be interested in. For example, if your application is a drawing application, you are almost certainly interested in the mouse movement message. If your window detects that the mouse is moving *and* the left mouse button is being pressed, your application might interpret that to mean “The user is trying to draw a line—paint a pixel at this point.”

Switch statements in a classic C/SDK program are often rather daunting. Normal C/SDK switch statements look something like Listing 3-1:

Listing 3-1. A typical C/SDK window procedure

```
LRESULT EXPORT WINAPI WndProc(HWND hwnd,
                           UINT message,
                           WPARAM wParam,
                           LPARAM lParam) {
    LONG lRet = 0L;

    switch (message) {
        case WM_CREATE:
            HandleCreate(hWnd, wParam, lParam);
            break;

        case WM_COMMAND:
            switch (wParam) {
                case IDM_ABOUT:
                    HandleAbout(hWnd);
                    break;

                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;

                default:
                    break;
            }
            break;
    }
}
```

```

case WM_PAINT: {
    PAINTSTRUCT ps;
    HDC         hdc;

    hdc = BeginPaint(hwnd, &ps);
    EndPaint(hwnd, &ps);
}
break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    lRet = DefWindowProc(hwnd, message, wParam, lParam);
    break;

return lRet;
}
}
}

```

Notice that the switch statement has several layers to it. For example, once you find out that you have a WM_COMMAND message, you need to examine the wParam to find out more information about the command.

Few people want to deal with a switch statement like this one. It quickly grows out of control. And when you want to add a new message, it's hard to figure out where to put the new case, especially when the switch statement spans several pages. In addition, missing a break statement can be disastrous.

That's how a C/SDK Windows program interprets messages. Let's see how you might handle messages in a C++ program.

Windows and C++

Consider how you might use C++ classes to handle Windows messages. When a C++ programmer hears the word *polymorphic*, he or she usually thinks of virtual functions right away. At first glance, handling window messages through virtual member functions makes sense. You just create a window base class and provide virtual member functions for each possible message the window might receive.

Though this method is consistent with C++ and OOP traditions, there is a problem with this approach. Remember, virtual functions are implemented through a virtual function table that accompanies a class. Moreover, every subsequent class derived from that class carries its own copy of the virtual function table. Each entry into the virtual function table (vtable) is a four-byte pointer. Count the number of messages that need to be handled by a virtual function and multiply that number by

four (for a four-byte pointer). Yikes!—each class ends up lugging around a huge number of extra bytes in pure vtable overhead.

There is another problem with implementing polymorphic behavior for a window through virtual functions. The number and kind of window messages may change from time to time. Using virtual functions as message handlers for these messages may cause code to break if and when the messages change.

To overcome the obstacles presented by handling messages through virtual functions, Microsoft developed a message routing and handling system that is efficient, extensible, and not compiler specific. The solutions they came up with are called message maps. At the highest level, message maps simply associate window messages and commands to a class's member functions. Here's the story behind message maps and how they work in your MFC-based application.

MFC Message-Mapping Internals

MFC's message mapping technology is made up of two parts: (1) the CCmdTarget class and (2) message maps. The CCmdTarget class is the base class for any object that needs to receive window messages, commands, or both. A message map is the mechanism that associates window messages to class member functions handling the message. Message map data structures and message map macros are two other important aspects of the message-mapping system.

The CCmdTarget Class

To be able to receive messages, a class must be derived from CCmdTarget. CCmdTarget-derived classes have the machinery necessary to deal with message maps. Any class derived from CCmdTarget can use a message map. When you skim through the MFC hierarchy charts, you notice that a number of classes are derived from CCmdTarget, including CWnd, CDocument, and CWinApp.

Message Map Data Structures

Before looking at real message maps, let's take a peek at two data structures used to implement MFC's message mapping. The first is AFX_MSGMAP_ENTRY. This structure represents the actual entries into the message map table:

```
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage;
```

```

    UINT nCode;
    UINT nID;
    UINT nLastID;
    UINT nSig;
    AFX_PMSG pfn;
};

}

```

The first field (*nMessage*) indicates the Windows message coming through the system. These are the same messages that appear in the big switch statement associated with C/SDK-based programs. The second field (*nCode*) represents the control code or the WM_NOTIFY code; this field is new with MFC 3.0. The third field (*nID*) refers to the control ID generating the message. Parameter number four (*nLastID*) is also new for MFC 3.0. It is used for entries specifying a range of control identifiers (also new for MFC 3.0). The *nSig* parameter indicates the signature of the function to handle the message (covered later in this chapter). The last parameter (*pfn*) points to the routine handling the message.

The second structure is AFX_MSGMAP. The AFX_MSGMAP structure represents the actual message map:

```

struct AFX_MSGMAP
{
    const AFX_MSGMAP* pBaseMap;
    const AFX_MSGMAP_ENTRY* lpEntries;
};

}

```

This structure has two parts: (1) a pointer to another AFX_MSGMAP structure (in practice, the base class's message map) and (2) an array of AFX_MSGMAP_ENTRY structures. Notice how this structure is set up to be included in a linked list. A message map is basically an array of AFX_MSGMAP_ENTRY structures. Each class hierarchy used within an application maintains a linked list of message maps. This is how MFC implements inheritance using message maps. If a message isn't handled by a class's own message map, the framework checks the base class's message map. Basically, the framework walks the message maps back to the root class until it finds a function to handle the message.

Message Map Macros

MFC provides three macros to generate message maps: DECLARE_MESSAGE_MAP, BEGIN_MESSAGE_MAP, and END_MESSAGE_MAP. These macros expand into code that defines and implements a message map for a CCmdTarget-based class.

When using message maps in your classes, the basic strategy is to include DECLARE_MESSAGE_MAP in your class definition (your H file) and then add

BEGIN_MESSAGE_MAP, END_MESSAGE_MAP, and the message-mapping information to your CPP file.

DECLARE_MESSAGE_MAP, defined in AFXWIN.H, looks like this:

```
#define DECLARE_MESSAGE_MAP() \
private: \
    static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
    static const AFX_MSGMAP messageMap; \
    virtual const AFX_MSGMAP* GetMessageMap() const;
```

Using DECLARE_MESSAGE_MAP in a class declaration defines three things for the class: (1) an array of AFX_MSGMAP_ENTRY structures called _messageEntries, (2) an AFX_MSGMAP structure called messageMap, and (3) a function to retrieve the class's message map (GetMessageMap()). Note that the message map entries (_messageEntries) and the message map structure (messageMap) are static members of the class. This means that there is one _messageEntries array and one messageMap member for all objects within the class.

Here is a CView-derived class called CTestView that includes a message map. The preprocessor uses the message map macros to generate message-mapping support code. Given this C++ code:

```
class CTestView : public CView
{
    ...
protected:
    //{{AFX_MSG(CTestView)
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

the preprocessor creates the following class definition:

```
class CTestView : public CView
{
    ...
protected:
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
private:
    static const AFX_MSGMAP_ENTRY _messageEntries[];
protected:
    static const AFX_MSGMAP messageMap;
    virtual const AFX_MSGMAP* GetMessageMap() const;
};
```

Once the definitions for the classes are done, there are two macros that finish the job: BEGIN_MESSAGE_MAP and END_MESSAGE_MAP.

As the name implies, the BEGIN_MESSAGE_MAP macro begins a message map. The macro definition, as found in AFXWIN.H, looks like this:

```
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
    const AFX_MSGMAP* theClass::GetMessageMap() const \
    { return &theClass::messageMap; } \
AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
    { &baseClass::messageMap, &theClass::_messageEntries[0] }; \
const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
{
```

The END_MESSAGE_MAP macro ends a message map. The definition is also found in AFXWIN.H:

```
#define END_MESSAGE_MAP() \
{ 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
};
```

When used together in an implementation file (a C++ file), these macros actually implement the message map. Given this code for the test view:

```
BEGIN_MESSAGE_MAP(CTestView, CView)
//{{AFX_MSG_MAP(CTestView)
ON_COMMAND(ID_STRING_CENTER, OnStringCenter)
ON_WM_LBUTTONDOWNDBLCLK()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

the preprocessor emits the following code:

```
const AFX_MSGMAP* CTestView::GetMessageMap() const
{ return &CTestView::messageMap; }

AFX_DATADEF const AFX_MSGMAP CTestView ::messageMap =
{ &CView::messageMap, &CTestView::_messageEntries[0] };

const AFX_MSGMAP_ENTRY CTestView::_messageEntries[] =
{
    ON_COMMAND(ID_STRING_CENTER, OnStringCenter)
    ON_WM_LBUTTONDOWNDBLCLK()
    { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
};
```

First, the macros generate a GetMessageMap() function for the class. This function simply returns a reference to the class's message map. GetMessageMap() is used by the framework to retrieve the class's message map whenever it needs to.

The macro then generates code that fills the class's AFX_MSGMAP structure. The first field points to the base class's message map (in this case the CView's message map), creating a linked list all the way back to the root object. This allows the framework to check each of the base classes for a message handler whenever a specific message handler can't be found. The linked list is how MFC implements inheritance in message maps. The second field is set to point to the first message map entry for the class itself.

Finally, the macro generates the actual message table. Remember, the message map is simply a table of AFX_MSGMAP_ENTRY structures.

Notice the additional macros enclosed within the message map. There are a number of these listed with the MFC documentation. They all begin with "ON_ . . ." These macros map window messages to their specific message handlers.

Sandwiched between the BEGIN_MESSAGE_MAP and END_MESSAGE_MAP macros is a set of these message map entry macros. The macros expand to fill the _messageEntries array for the class's message map. MFC defines various message map entry macros, which are listed in Table 3-1.

In the specific example just given, the message map uses two of these macros: ON_COMMAND and ON_WM_LBUTTONDOWNDBLCLK. They are found in AFXMSG.H, and they appear in Listing 3-2.

Listing 3-2. The ON_COMMAND and ON_WM_LBUTTONDOWNDBLCLK macros, from AFXMSG.H

```
ONCOMMAND macro:
#define ON_COMMAND(id, memberFxn) \
{ WM_COMMAND, \
CN_COMMAND, \
(WORD)id, \
(WORD)id, \
AfxSig_vv, \
(AFX_PMSG)memberFxn },

ON_WM_LBUTTONDOWNDBLCLK macro:
#define ON_WM_LBUTTONDOWNDBLCLK() \
{ WM_LBUTTONDOWNDBLCLK,
0, \
0, \
0, \
AfxSig_vwp, \
(AFX_PMSG)(AFX_PMSGW) \
(void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))OnLButtonDblClk },
```

Table 3-1. MFC's message map entry macros

Message Type	Macro Form	Arguments
Predefined Windows messages	ON_WM_XXXX	None
Commands	ON_COMMAND	Command ID, Handler name
Update commands	ON_UPDATE_COMMAND_UI	Command ID, Handler name
Control notifications	ON_XXXX	Control ID, Handler name
User-defined message	ON_MESSAGE	User-defined message ID, Handler name
Registered Windows message	ON_REGISTERED_MESSAGE	Registered message ID variable, Handler name
Range of command IDs	ON_COMMAND_RANGE	Start and end of a contiguous range of command IDs
Range of command IDs for updating	ON_UPDATE_COMMAND_UI_RANGE	Start and end of a contiguous range of command IDs
Range of control IDs	ON_CONTROL_RANGE	A control-notification code and the start and end of a contiguous range of command IDs

Go back and take a quick look at the AFX_MSGMAP_ENTRY structure. Notice how the macro neatly fills the structure. Note that the fifth field specifies the signature of the function used to handle the message. The signature values are used by the framework to signify the return type and the parameters of the message-handling function.

How MFC Uses Message Maps

So, now you've seen how the message map is built. Let's take a look at how it's used.

An MFC-based program deals with two kinds of messages: (1) regular window messages (like WM_MOUSEMOVE) and (2) commands (that is, the messages generated from menus and controls and represented by the WM_COMMAND message). Message maps handle both kinds of messages.

The best way to understand the message-mapping architecture is to follow a couple of window messages through an application. First, we'll follow a WM_COMMAND message through the application framework to see where it goes. Then we'll watch a regular window message as it travels through the framework. We'll see where a window message starts its journey through the framework and where it stops.

How MFC Windows Become Wired to a WndProc

In the 16-bit version of MFC (version 2.5), MFC registered a single message-handling procedure called AfxWndProc() for all its windows (except for the non-MFC windows registered by the developer). As with any other Windows application, an MFC application's messages are deposited in a window procedure. In the case of the 16-bit version of an MFC application, that function was AfxWndProc(). Because the MFC window classes were registered with AfxWndProc() as the message handler, it was obvious where messages were handled. Following messages through the system was easy: you just put a breakpoint in the AfxWndProc() function.

Things have changed significantly with the 32-bit version of MFC. MFC window classes are no longer registered with the AfxWndProc() function—they're registered with DefWindowProc() as the message handler. When you look at the source code for MFC, you'll find that AfxWndProc() is still there, and it's dispatching messages to various CWnd-derived objects. How is it that messages end up in AfxWndProc()?

Remember that MFC maintains two of its own message hooks: _AfxMsgFilterHook() and _AfxCbtFilterHook(). Because messages are directed to the hooks before anything else happens, there's an opportunity for certain messages to be intercepted in one of these hooks. As it turns out, MFC uses the computer-based training hook function to attach AfxWndProc() to the MFC windows as they're created. Here's how AfxWndProc() is hooked up to MFC's windows.

MFC installs the _AfxCbtFilterHook() function whenever a new CWnd-derived object is created—that is, during a call to CWnd::CreateEx(). Right before CWnd::CreateEx() makes a call to the CreateWindowEx() API function, CWnd::CreateEx() calls AfxHookWindowCreate(). AfxHookWindowCreate() inserts _AfxCbtFilterHook() into the hook chain. Because _AfxCbtFilterHook() is a computer-based training hook, Windows calls _AfxCbtFilterHook() before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window. Windows also calls _AfxCbtFilterHook before completing a system command, before removing a mouse or keyboard event from the system message queue, before setting the keyboard focus, and before synchronizing with the system message queue.

_AfxCbtFilterHook() sits there receiving the messages as just described. _AfxCbtFilterHook() ignores window messages until the HCBT_CREATEWND

code is passed into `_AfxCbtFilterHook()`. When `_AfxCbtFilterHook()` is called with the `HCBT_CREATEWND` code, that means a window is about to be created. `_AfxCbtFilterHook()` then calls `_AfxStandardSubclass()`.

`_AfxStandardSubclass()` uses `SetWindowLong()` to wire `AfxWndProc()` up to the window. From now on, messages for that window will go to `AfxWndProc()`, where they are handled by the command-routing architecture. So even though the window was originally registered with `DefWindowProc()` as the message-handling procedure, the framework effectively wires the windows up to `AfxWndProc()` whenever a `CWnd`-derived window is created.

The main reason Microsoft shifted from using `AfxWndProc()` as the registered window procedure to using `DefWindowProc()` is to support 3D Controls, which works through Microsoft's `CTL3D.DLL`. It is desirable to have `CTL3D` functionality appear to be part of the system. This is so an app that wants to override `CTL3D` functionality (in handling `WM_CTLCOLOR` messages, primarily) can do so. In order for this to work, MFC has to ensure that subclassing is in the following order (from first called to last called): `AfxWndProc()`, `CTL3D`'s `WndProc()`, and finally `DefWindowProc()`. In order to do this, Microsoft had to allow `CTL3D` to subclass *before* `AfxWndProc()`, which meant delaying hooking up `AfxWndProc()` until after `CTL3DSubclassDlgEx` is called. So MFC registers everything with `DefWindowProc()`, then during `_AfxStandardSubclass()`, hooks up `CTL3D`, then finally adds a subclass on top of `CTL3D` by installing `AfxWndProc()` as the final "top" window procedure. For backward compatibility Microsoft still supports the idea of registering with `AfxWndProc()` from the start (you'll see extra tests in `_AfxStandardSubclass()` to handle this).

Handling Messages

Remember that MFC represents windows in two ways: (1) by a unique system-defined window handle and (2) by the C++ class representing the window. Window handles are wrapped by `CWnd` and `CWnd`-derived classes. It's easy to get the window handle from a `CWnd` object because the window handle is a data member of the class. However, there is no way to get from the window handle to the `CWnd` object without some extra footwork.

As you saw earlier, MFC uses a `CMapPtrToPtr` object to map handles to `CWnd` objects. MFC maintains the link for the lifetime of the window. When a window is created using `CWnd` (or a `CWnd`-derived class), the window handle is attached to the `CWnd` object. That is, the window handle and the `CWnd` object are associated through the handle map. MFC does this so that the framework code can deal with C++ objects rather than window handles.

And now back to the window procedure. `AfxWndProc()` handles a single specific message: `WM_QUERYAFXWNDPROC`. If the incoming message is

WM_QUERYAFXWNDPROC, AfxWndProc() returns the number 1. Applications can send the WM_QUERYAFXWNDPROC message to find out if the window is an MFC window using MFC's message-mapping system. The return value for the WM_QUERYAFXWNDPROC message is also used by CWnd's Dump procedure for diagnostic information. If the message isn't WM_QUERYAFXWNDPROC, AfxWndProc() goes on to process the message.

Inside AfxWndProc(), the framework retrieves the C++ object associated with the window handle using CWnd::FromHandlePermanent(). The framework then calls AfxCallWndProc(). Notice that even though the first parameter is a pointer to a CWnd object, the second parameter is a window handle. This allows AfxCallWndProc() to maintain a record of the last message processed for use in handling exceptions and debugging. Here's the prototype for AfxCallWndProc(). Notice how it looks like any other window procedure, except that the parameter includes a CWnd pointer as well.

```
LRESULT PASCAL _AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT message,
                                WPARAM wParam, LPARAM lParam);
```

AfxCallWndProc() first examines the message to see if it's a WM_INITDIALOG message, in which case it calls _AfxHandleInitDialog(). _AfxHandleInitDialog() is for the auto-center dialog feature. MFC caches certain styles before the dialog handles WM_INITDIALOG. If it is appropriate to center the window (the window is still not visible and hasn't moved), then MFC automatically centers the dialog against its parent. The function AfxCallWndProc() saves the window handle, the message, and the WPARAM and the LPARAM in the current thread state's m_lastSentMsg member variable. Then AfxCallWndProc() simply calls the window object's window procedure. Here's the prototype for WindowProc():

```
LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam);
```

Notice that the signature has the same parameters as a regular window procedure. That's because at this point, MFC has already mapped the window handle to the existing CWnd-derived class.

By the way, CWnd::WindowProc() is virtual, so you can override it if you'd like. One reason you may want to override CWnd::WindowProc() is if you want to handle a message before MFC even looks at it. For example, you may have a class that bypasses the message-mapping system—perhaps to improve performance.

CWnd::WindowProc() calls CWnd::OnWndMsg(). If OnWndMsg() returns FALSE, then CWnd::WindowProc() calls CWnd::DefWindowProc().

The action really begins inside CWnd::OnWndMsg(). Listing 3-3 shows some pared-down source code.

Listing 3-3. The CWnd::OnWndMsg() abbreviated code

```

BOOL CWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult) {
    if (message == WM_COMMAND)
        call OnCommand(wParam, lParam) and return;

    if (message == WM_NOTIFY)
        call OnNotify(wParam, lParam, &lResult) and return;

    if (message == WM_ACTIVATE)
        call _AfxHandleActivate(this, wParam,
                               CWnd::FromHandle((HWND)lParam)) and return;

    if (message == WM_SETCURSOR) &&
        call _AfxHandleSetCursor(this,
                               (short)LOWORD(lParam),
                               HIWORD(lParam))) and return;

    // Find the message map entry in a message cache using its hash value

    if (the message is in the cache ) {
        if (message map entry is NULL)
            return FALSE;

        if (message < 0xC000) // Registered Windows message
            goto LDispatch; // Call the function
        else
            goto LDispatchRegistered; // Call the function for a registered
                                         // message
    } else {
        // not in cache, look for it
        msgCache.nMsg = message;
        msgCache.pMessageMap = pMessageMap;

        for /* pMessageMap already init'ed */; pMessageMap != NULL;
            pMessageMap = pMessageMap->pBaseMap) {

            if (message < 0xC000) {
                // constant window message
                if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
                                                   message, 0, 0)) != NULL) {
                    msgCache.lpEntry = lpEntry;
                    goto LDispatch;
                }
            } else {
                // registered windows message
                lpEntry = pMessageMap->lpEntries;
            }
        }
    }
}

```

```

        while ((lpEntry = AfxFindMessageEntry(lpEntry, 0xC000, 0, 0))
               != NULL) {
            UINT* pnID = (UINT*)(lpEntry->nSig);
            // must be successfully registered
            if (*pnID == message) {
                msgCache.lpEntry = lpEntry;
                goto LDispatchRegistered;
            }
            lpEntry++;      // keep looking past this one
        }
    }

msgCache.lpEntry = NULL;
return FALSE;
}

LDispatch:
ASSERT(message < 0xC000);
union MessageMapFunctions mmf;
mmf.pfn = lpEntry->pfn;

switch (lpEntry->nSig) {
default:
    ASSERT(FALSE);
    break;

// Examine the signature type, calling the
// message handler with the appropriate parameters
}
goto LReturnTrue;

LDispatchRegistered: // for registered windows messages
ASSERT(message >= 0xC000);
mmf.pfn = lpEntry->pfn;
lResult = (this->*mmf.pfn_lwl)(wParam, lParam);

LReturnTrue:
if (pResult != NULL)
    *pResult = lResult;
return TRUE;
}

```

This is a fairly hefty function. Remember, we're replacing that unruly switch statement. Let's briefly walk through OnWndMsg() before tracing messages through it. First, OnWndMsg() tries to filter out certain messages from the get-go: WM_COMMAND, WM_NOTIFY, WM_ACTIVATE, and WM_SETCURSOR. The framework has special ways of handling each of these messages. If the message isn't one of those

just listed, `OnWndMsg()` tries to look up the message in the message map. MFC keeps a message map entry cache that is accessible via a hash value. This is a great optimization because looking up a value in a hash table is *much* cheaper than walking the message map.

This is where commands and regular window messages go their separate ways. If the message is a command message (that is, `nMsg == WM_COMMAND`), then `CWnd::OnWndMsg()` calls `OnCommand()`. Otherwise, it retrieves the window object's message map to process the message (more on that shortly). Let's examine the command routing first.

Handling WM_COMMAND

The first stop a command makes on its way to its designated command target is `CWnd::OnCommand()`.

CWnd::OnCommand()

`OnCommand()` is a virtual function, so the framework calls the correct version. In this example, because the message was generated for the main frame window, the framework calls the `CFrameWnd` version of `OnCommand()`.

```
BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam);
```

By this point, the message is pared down to just its WPARAM and its LPARAM. If the message is a request for on-line help, the framework sends a WM_COMMANDHELP message to the frame window. Otherwise, the message is passed on to the base class's OnCommand(), CWnd::OnCommand().

`OnCommand()` examines the LPARAM, which represents the control. If the command was generated by a control, then the LPARAM contains the control window. If the message is a control notification (like LBN_CHANGESEL), then the framework performs some special processing. If the message is for a control, `OnCommand()` sends the last message straight to the control, then `OnCommand()` returns.

Otherwise, CWnd::OnCommand() makes sure that the user-interface element that generated the command has not become disabled (for instance, a menu item) and passes the message on to OnCmdMsg() (which is also virtual). Again, because the frame window is still trying to handle the message, CFrameWnd::OnCmdMsg() is the version that's called. This function is found in WINFRM.CPP:

CFrameWnd::OnCmdMsg() passes NULL for pExtra and pHandlerInfo when it calls CFrameWnd::OnCmdMsg(), because this information is not needed for handling commands.

CFrameWnd::OnCmdMsg() pumps the message through the application components in this order:

- The active view
- The active view's document
- The main frame window
- The application

To route the command to the active view, CFrameWnd::OnCmdMsg() tries to find the frame's active view using CWnd::GetActiveView(). If CFrameWnd::OnCmdMsg() succeeds in finding the frame's active window, it calls the active view's OnCmdMsg(). If the active view's OnCmdMsg() can't deal with the command, the document gets a crack at the command. If CFrameWnd::OnCmdMsg() fails to find an active view, or the view and the document fail to handle the message, then the frame window gets a chance to handle the message. Finally, if the frame window doesn't want the message, then the application takes a crack at the message—CFrameWnd::OnCmdMsg() calls the application's OnCmdMsg() function.

At this point in the example, the message has reached the active view and the function CView::OnCmdMsg(), found in VIEWCORE.CPP:

```
BOOL CView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                      AFX_CMDHANDLERINFO* pHandlerInfo);
```

The framework gives the window pane part of the view a chance to respond to the message by calling CWnd::OnCmdMsg(). If the view pane can't handle the message, it's pumped through the document.

Because CWnd doesn't override OnCmdMsg(), the command goes straight to CCmdTarget::OnCmdMsg(), found in CMDTARG.CPP:

```
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                           AFX_CMDHANDLERINFO* pHandlerInfo);
```

CCmdTarget::OnCmdMsg() walks the message map (back to the base class if it has to) trying to find a handler for the message. If it finds one, it calls that function. If it can't, CCmdTarget::OnCmdMsg() returns FALSE, and the document gets a chance to handle the message. If the document doesn't want anything to do with the message, then the message is handled by the CWnd's DefWindowProc().

If CCmdTarget::OnCmdMsg() finds a handler in the message map, then it calls DispatchCmdMsg(), also in CMDTARG.CPP:

```
static BOOL DispatchCmdMsg(CCmdTarget* pTarget, UINT nID, int nCode,
                           AFX_PMSG pfn, void* pExtra, UINT nSig,
                           AFX_CMDHANDLERINFO* pHandlerInfo);
```

This function is declared static, so it's visible only within CMDTARG.CPP. One of the parameters is the function signature. This signature comes from the message map entry itself. (Remember, one of the fields of the message map structure is a function signature.) Also notice that a pointer to the message handler function (pfn) is passed as a parameter. This function pointer is also found within the message map entry.

DispatchCmdMsg() switches on the function signature, performing different operations depending on whether the signature is for a regular command, an extended command, a command user-interface handler, or a Visual Basic control. In the case of a regular menu command, the signature is AfxSig_vv (void return, void parameter list). DispatchCmdMsg() immediately calls the message handler, and—voilà—the handler for that message is called.

If CCmdTarget::OnCmdMsg() fails to find a handler within the message map, it returns FALSE, which eventually causes CWnd::DefWindowProc() to handle the message.

Table 3-2 shows the various MFC function signatures and their parameters and return values.

Table 3-2. Function signatures used in message mapping

Signature Name	Return Type	Parameters
AfxSig_bD	BOOL	CDC*
AfxSig_bb	BOOL	BOOL
AfxSig_bWww	BOOL	CWnd*, UINT, UINT
AfxSig_hDWww	HBRUSH	CDC*, CWnd*, UINT
AfxSig_iWwww	int	UINT, CWnd*, UINT
AfxSig_iWww	int	CWnd*, UINT, UINT
AfxSig_is	int	LPTSTR
AfxSig_lwl	LRESULT	WPARAM, LPARAM
AfxSig_lwwM	LRESULT	UINT, UINT, CMenu*
AfxSig_vv	void	void

(continued)

Table 3-2. Function signatures used in message mapping (continued)

<i>Signature Name</i>	<i>Return Type</i>	<i>Parameters</i>
AfxSig_vw	void	UINT
AfxSig_vww	void	UINT, UINT
AfxSig_vvii	void	int, int (wParam is ignored)
AfxSig_vwww	void	UINT, UINT, UINT
AfxSig_vvii	void	UINT, int, int
AfxSig_vwl	void	UINT, LPARAM
AfxSig_vbWW	void	BOOL, CWnd*, CWnd*
AfxSig_vD	void	CDC*
AfxSig_vM	void	CMenu*
AfxSig_vMwb	void	CMenu*, UINT, BOOL
AfxSig_vW	void	CWnd*
AfxSig_vWww	void	CWnd*, UINT, UINT
AfxSig_vWh	void	CWnd*, HANDLE
AfxSig_vwW	void	UINT, CWnd*
AfxSig_vwWb	void	UINT, CWnd*, BOOL
AfxSig_vwwW	void	UINT, UINT, CWnd*
AfxSig_vs	void	LPTSTR
AfxSig_vOWNER	void	int, LPTSTR (force return TRUE)
AfxSig_iis	int	int, LPTSTR
AfxSig_wp	UINT	CPoint
AfxSig_wv	UINT	void
AfxSig_vPOS	void	WINDOWPOS*
AfxSig_vCALC	void	NCCALCSIZE_PARAMS*
AfxSig_vNMHDRpl	void	NMHDR*, LRESULT*
AfxSig_vwNMHDRpl	void	UINT, NMHDR*, LRESULT*
AfxSig_bwNMHDRpl	BOOL	UINT, NMHDR*, LRESULT*
Signatures specific to CCmdTarget		
AfxSig_cmdui	void	CCmdUI*
AfxSig_cmduiw	void	CCmdUI*, UINT
AfxSig_vp	void	void*
AfxSig_bp	BOOL	void*
Other aliases based on implementation		
AfxSig_vwwh	void	UINT, UINT, HANDLE
AfxSig_vwp	void	UINT, CPoint
AfxSig_bw = AfxSig_bb	BOOL	UINT

Table 3-2. Function signatures used in message mapping (continued)

Signature Name	Return Type	Parameters
AfxSig_bh = AfxSig_bb	BOOL	HANDLE
AfxSig_iw = AfxSig_bb	int	UINT
AfxSig_ww = AfxSig_bb	UINT	UINT
AfxSig_bv = AfxSig_ww	BOOL	void
AfxSig_hv = AfxSig_ww	HANDLE	void
AfxSig_vb = AfxSig_ww	void	BOOL
AfxSig_vbh = AfxSig_www	void	BOOL, HANDLE
AfxSig_vbw = AfxSig_www	void	BOOL, UINT
AfxSig_vhh = AfxSig_www	void	HANDLE, HANDLE
AfxSig_vh = AfxSig_ww	void	HANDLE
AfxSig_end = 0	[marks end of message map]	

MFC uses this command-routing scheme for all CCmdTarget-derived classes. That includes classes derived from CWnd, CDocument, CView, and CFrameWnd. One interesting aspect of this arrangement is the path that commands take to get to their final destinations. All command messages take the same path for the first three steps. That is, the message first lands in AfxWndProc(), which gets the CWnd object from the HWND parameter and calls AfxCallWndProc(). AfxCallWndProc() calls the CWnd-derived object's WindowProc(). From there, the message is routed to its intended destination. Here's a rundown of the path a command message takes to the various components of an MFC application.

Command to a Frame Window

Here is the path a WM_COMMAND message takes to an application's frame window. As with all Windows messages through an MFC program, the first stop is AfxWndProc(). This calls AfxCallWndProc(), finally ending up in the specific window's window procedure. From there, the command message is routed to the appropriate command target.

```

AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CMainFrame::OnFrameAframecommand()

```

Command to a Document

Here is the path a WM_COMMAND message takes to an application's document:

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CView::OnCmdMsg()
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CSdiappDoc::OnDoccommandAdoccommand()
```

Command to a View

Here is the path a WM_COMMAND message takes to an application's view:

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CSdiappView::OnViewAviewcommand()
```

Command to an App

Here is the path a WM_COMMAND message takes to an application's CWinApp-derived object:

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CSdiappApp::OnAppAnappcommand()
```

Command to a Dialog Box

Dialog boxes also receive command messages. Here is the path a WM_COMMAND message takes to a dialog box:

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CWnd::OnCommand()
CDialog::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CAboutDlg::OnAButton()
```

So that's how command messages come through the framework. As you can see, the message goes caroming between several different classes. Handling regular window messages (like WM_SIZE) is quite a bit simpler.

Handling Regular Window Messages

As with command messages, a regular message starts out in AfxWndProc(), which goes on to turn the window handle into a CWnd object. AfxWndProc() then calls AfxCallWndProc(), which in turn calls the CWnd object's WindowProc(). WindowProc() then calls OnWndMsg(), which calls AfxFindMessageEntry() to find a handler for the message.

AfxFindMessageEntry()

AfxFindMessageEntry() is an interesting function. There are actually two versions of it: one written in assembly language and one written in C. If the application is compiled for an Intel-based machine, then AfxFindMessageEntry() uses the assembly version. Otherwise, AfxFindMessageEntry() uses the C version.

To find the message map entry in the table, OnWndMsg() first retrieves the CWnd object's message map. Then, given the first entry in that message map, AfxFindMessageEntry() walks the array of message map entries until it either (a) finds the message in the message map or (b) finds the end of the message map as marked by the AfxSig_end signature (remember, that's what the END_MESSAGE_MAP macro does).

When OnWndMsg() finds a handler, it calls the handler. If the message map doesn't include a handler for a specific message, the framework calls the window's default window procedure. So, unlike commands, which are routed to several places, regular window messages go straight to the window for which they were intended. For example, take a look at the path a WM_SIZE message takes to its destination (a CWnd-derived class).

Sending a WM_SIZE Message to a View

Here is the path that a WM_SIZE message takes to an application's view. Notice how window messages make a beeline to their CWnd-derived objects.

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CSdiappView::OnSize()
```

Calling the Member Function

Before leaving message maps, let's take a look at how MFC calls the member function for a particular window message once it finds the entry in the message map. Recall that one of the members of the message map entry structure is a pointer to the function handling the window message. To deal with this, OnWndMsg() declares a MessageMapFunctions union on the stack. Listing 3-4 shows a portion of the union:

Listing 3-4. Signatures for MFC's message-handler functions

```
union MessageMapFunctions
{
    AFX_PMSG pfn;      // generic member function pointer

    // specific type safe variants
    BOOL      (AFX_MSG_CALL CWnd::*pfn_bD)(CDC* );
    BOOL      (AFX_MSG_CALL CWnd::*pfn_bb)(BOOL );
    BOOL      (AFX_MSG_CALL CWnd::*pfn_bWw)(CWnd*, UINT, UINT);
    BOOL      (AFX_MSG_CALL CWnd::*pfn_bHelpInfo)(HELPINFO* );
    HBRUSH   (AFX_MSG_CALL CWnd::*pfn_hDwW)(CDC*, CWnd*, UINT);
    HBRUSH   (AFX_MSG_CALL CWnd::*pfn_hDw)(CDC*, UINT);
    int      (AFX_MSG_CALL CWnd::*pfn_iwWw)(UINT, CWnd*, UINT);
    int      (AFX_MSG_CALL CWnd::*pfn_iWW)(UINT, UINT);
    int      (AFX_MSG_CALL CWnd::*pfn_iWWw)(CWnd*, UINT, UINT);
    int      (AFX_MSG_CALL CWnd::*pfn_is)(LPTSTR );
    LRESULT  (AFX_MSG_CALL CWnd::*pfn_lwl)(WPARAM, LPARAM );
    LRESULT  (AFX_MSG_CALL CWnd::*pfn_lwwM)(UINT, UINT, CMenu* );
    void     (AFX_MSG_CALL CWnd::*pfn_vv)(void );

    void     (AFX_MSG_CALL CWnd::*pfn_vw)(UINT );
    void     (AFX_MSG_CALL CWnd::*pfn_vww)(UINT, UINT );
    void     (AFX_MSG_CALL CWnd::*pfn_vwii)(UINT, int, int );

    ...
};
```

The MessageMapFunctions union contains a generic AFX function pointer as a member, followed by prototypes for all the different kinds of functions that might

be used for a message handler. OnWndMsg() sets the MessageMapFunctions union to the message handler's address:

```
mmf.pfn = lpEntry->pfn;
```

Next, OnWndMsg() goes on to find the signature that matches the signature for a WM_SIZE message. Once OnWndMsg() finds a match, it pulls the necessary parameters from the WPARAM and the LPARAM and calls the handler using the prototype included in the MessageMapFunctions structure.

For example, here's the line of code that executes a handler for WM_SIZE.

```
case AfxSig_vwii:
    (this->*mmf.pfn_vwii)(wParam, LOWORD(lParam), HIWORD(lParam));
    break;
```

Pretty slick, huh? Some have said looking at MFC is kind of like touring a sausage factory. The sausages taste good and work fine (as far as sausages go), but you really should never look at how they make the sausages. MFC's message-mapping code is sometimes a little hard to follow. However, it does work very well and it is an important component of Visual C++'s Wizard technology.

In addition to WM_COMMAND and the regular window messages, MFC handles certain other window messages in special ways.

Other Kinds of Messages

The WM_COMMAND and various window messages (like WM_SIZE and WM_MOVE) are the most common kinds of messages your app will receive. However, MFC has special ways to handle the WM_NOTIFY, the WM_ACTIVATE, and the WM_SETCURSOR messages. Let's take a brief look at these special cases.

WM_NOTIFY and Message Reflection

In earlier versions of Windows, WM_COMMAND was an overloaded message that could represent (1) a command from a menu item or (2) a notification from a child window. With the advent of Windows 95 and the new common controls, there's a new message called WM_NOTIFY. WM_NOTIFY is a generalized control notification. WM_NOTIFY is always a notification, whereas WM_COMMAND can be either a command or a notification.

CWnd::OnWndMsg() handles WM_NOTIFY specifically through the CWnd::OnNotify() member function. Instead of passing the regular WPARAM and LPARAM arguments, WM_NOTIFY packs a NMHDR structure in the LPARAM:

```

struct NMHDR {
    HWND hwndFrom;      // control that sent notification
    UINT idFrom;        // ID of control
    UINT code;          // notification code
};


```

OnNotify() uses the hwndFrom field and ends up calling the virtual CWnd function OnChildNotify(). OnChildNotify() then sends the message directly to the control so that the control can deal with it (after all, it really is the control's responsibility).

In addition to handling WM_NOTIFY for the new controls, MFC provides a new mechanism called *message reflection*. Recall how control notifications worked in earlier versions of MFC: Whenever something interesting happened to the control, the control notified its parent so that the parent could do something about it. A good example is coloring a control in a dialog box. You could plant regular controls in a dialog box. However, if you wanted to change their colors, it was the dialog's responsibility to respond to the WM_CTLCOLOR message, coloring the controls as appropriate. Of course, this isn't the best way to do this sort of thing because it's not very self-contained. In an ideal world, the control would be responsible for painting itself. Message reflection provides this capability in MFC 4.0.

MFC defines some new message map macros for handling reflected messages. They are similar to the macros for the standard message map macros except that they have the label "REFLECT" attached to them. For example, the standard handler for the WM_CTLCOLOR message is ON_WM_CTLCOLOR. The ON_WM_CTLCOLOR takes a function that takes a pointer to a CDC and an unsigned integer as parameters and returns an HBRUSH. Adding this macro to the message map causes the control to be notified whenever the parent receives the WM_CTLCOLOR message. That way, the control can handle the message rather than depend on the parent to deal with the repainting.

WM_ACTIVATE

MFC's messaging architecture also handles WM_ACTIVATE. OnWndMsg() calls the non-C++ member function _AfxHandleActivate(). This function checks to see if the WM_ACTIVATE message is for the top-level window. If it is, _AfxHandleActivate() sends the WM_ACTIVATETOPLEVEL message to the top-level window.

WM_SETCURSOR

MFC handles the WM_SETCURSOR message within the function _AfxHandleSetCursor(). _AfxHandleSetCursor() checks to see if one of the mouse buttons is pressed. If a mouse button is down, _AfxHandleSetCursor() activates the last active window. This is necessary to work around a Windows bug. If a window is disabled and a modal dialog is parented to that disabled window, the app will not activate itself correctly

when the user clicks on the disabled window. Instead, the user must click directly on the dialog itself. If the dialog is small, it is probably covered. Normally, clicking on the disabled window would result in just a beep from DefWindowProc() in this circumstance. MFC fixes this Windows bug by activating the dialog box (and thus bringing the app to the top) when you click on a disabled window with an active modal dialog box (or other modal window).

Hooking into the Message Loop: PreTranslateMessage()

MFC has another useful feature: you can plug into the message loop to handle messages before they ever get to their designated command targets or windows. The function for doing that is PreTranslateMessage(). MFC gives you two places where you can hook into the message loop: CWinApp::PreTranslateMessage() and CWnd::PreTranslateMessage(). CWinApp::PreTranslateMessage() lets you process window messages even before they get to any of your application's windows or command targets.

CWinApp::Run() calls CWinApp::PreTranslateMessage() before the message is processed by TranslateMessage() and DispatchMessage(). CWndApp::PreTranslateMessage() takes a single parameter: a pointer to an MSG structure. By default, the framework examines the message. If the message is a mouse button down or a mouse button double-click, CWndApp::PreTranslateMessage() removes any tool tips from the screen by calling CancelToolTip().

Then CWinApp::PreTranslateMessage() calls each of the windows from the target window (as designated by the window handle in the message structure) to the application's main window. CWnd::WalkPreTranslateTree() performs this function, as shown in Listing 3-5.

Listing 3-5. CWnd's WalkPreTranslateTree() function

```
BOOL PASCAL CWnd::WalkPreTranslateTree(HWND hWndStop, MSG* pMsg)
{
    ASSERT(hWndStop == NULL || ::IsWindow(hWndStop));
    ASSERT(pMsg != NULL);

    // walk from the target window up to the hWndStop window checking
    // if any window wants to translate this message

    for (HWND hWnd = pMsg->hwnd; hWnd != NULL; hWnd = ::GetParent(hWnd))
    {
        CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
        if (pWnd != NULL)
```

```

{
    // target window is a C++ window
    if (pWnd->PreTranslateMessage(pMsg))
        return TRUE; // trapped by target window (eg: accelerators)
}

// got to hWndStop window without interest
if (hWnd == hWndStop)
    break;

}
return FALSE;      // no special processing
}

```

Before calling WalkPreTranslateTree(), CWinApp::PreTranslateMessage() retrieves the window handle of the application's main window by calling AfxGetMainWnd(). WalkPreTranslateTree() uses this handle to know when to stop calling PreTranslateMessage(). Then starting inside-out (that is, starting from the target window), WalkPreTranslateTree() hits each window, trying to find one interested in the message. For each window in the tree, WalkPreTranslateTree() fetches the CWnd object from the window handle map, calling PreTranslateMessage() for each CWnd object. WalkPreTranslateTree() does this for each window until either (1) the window handles the message or (2) WalkPreTranslateTree() reaches the application's main window.

CWnd::PreTranslateMessage() takes a single parameter: a pointer to the MSG structure. When you override PreTranslateMessage(), just examine the MSG structure. If you're interested in the message, go ahead and handle it within PreTranslateMessage() and return TRUE.

CWinApp::PreTranslateMessage() returns TRUE if it finds a window interested in the message. If CWinApp::PreTranslateMessage() returns TRUE, then CWinApp's message pump doesn't dispatch the message. So, PreTranslateMessage() effectively eats the message.

Conclusion

MFC's standard message-mapping technology is a reasonable alternative to handling messages via virtual class member functions. It eliminates the baggage of enormous vtables, it's fairly efficient, and it's compiler independent.

You should have a good grasp of how MFC handles the application (initialization and message pump) and window (message handling) aspects of a Windows application. Now we can start digesting the other various components of the framework. Let's start with MFC's utility classes.

The MFC Utility Classes

The MFC utility classes don't do any drawing or provide any grandiose application features. However, they are useful to you as an MFC programmer. Seeing how these classes work might not seem valuable at the outset. However, because many of the classes you write probably fall into the category of utility classes, learning the techniques that the MFC team uses in their utility classes will help you write better utility classes for your own projects.

In this chapter, we'll take a look first inside the simple value classes, such as `CString`, `CTime`, and `CTimeSpan`, and Windows structure wrappers such as `CPoint`, `CSize`, and `CRect`. After seeing how MFC implements the simple value classes, we'll examine the various collection classes and see what makes them tick. Next, we will dissect the `CFile` file utility class and its many derivatives. Finally, we will see how MFC provides advanced error-handling support through the class `CException` and its derivatives.

Along the way we'll be sure to point out any interesting techniques and previously undocumented information.

Simple Value Types

The simple value type utility classes encapsulate existing C++ types and Windows types such as `char *` and the `RECT` structure. In addition to providing C++ interfaces to the types that they wrap, these classes add extended features to them.

The simple value type classes provide C++ operators that make manipulating the types safer and easier. Some of the value type classes also enhance the basic type by adding memory management, internationalization, and other helpful member functions.

The most comprehensive example of a class that encapsulates a basic type is `CString`. Let's take a look at `CString` first, followed by some of the other simple value types.

Class `CString`: A `char *` on Steroids

`CString` encapsulates a character string and provides several other advanced features, including these:

- Internationalization
- Memory allocation/deallocation
- Operators (such as `+`, `=`, and `==`)
- String searching
- String manipulation (such as reverse and concat)
- Formatting (`printf`-like output)

Using a `CString`

Before looking at how MFC implements `CString`, let's review how to use the various features found in class `CString`.

`CString` provides a number of constructors that let you create a `CString` from a `char *`, from another `CString`, and even from a single character that you would like to have repeated. Here are some sample `CString` constructions:

```
//Create a CString from a char *
CString myString1("This is a string");
//Create a CString from a CString
CString myString2(myString1);
//Create a string with 80 Z characters
CString myString3('Z',80);
//Create an empty CString
CString myString4;
```

`CString` Operators

`CString` operators let you add, compare, assign, and even access characters in a string. Listing 4-1 shows some examples of `CString` operators:

Listing 4-1. Some example `CString` listings

```
CString myString1("string 1");
CString myString2("string 2");
CString myString3("string 3");
```

```

// ** Addition (concatenation) and assignment operators **
myString1 += myString2;
// result is "string 1string2" in myString1

myString3 = "This is " + myString2;
// result is "This is string 2" in myString3

myString2 += 'Z';
// result is "string 2Z" in myString2

// ** Comparison operators **
if (myString1 == myString2) // Like strcmp
... // Do something

if (myString1 < myString2)
... // Do something

if (myString1 != myString2)
... // Do something

// ** Operator [] (access char in string) **
// Get the eighth character (0 based array)
if (myString1[7] == '1')
... // Do something

```

CString provides many member functions for searching the string. Here's a summary of the searching member functions:

- **Find()**—Searches for a substring or character in the string. Returns index into string if found, -1 if not.
- **FindOneOf()**—Takes a char * and searches for the first character of both the argument and the CString. Returns the index if one is found, 1 if not.
- **ReverseFind()**—The same as find, but starts at the right of a string and works toward the left. Returns the index if found, -1 if not.

CString provides a Format() member function that takes printf-style variable arguments and places the results into the CString.

For example, the following code segment:

```
CString myString1; myString1.Format("There are %d at %x !",1000,0xffff);
```

yields “There are 1000 at 0xffff inside myString1.”

You can cut, make uppercase, make lowercase, grab subsections, and perform a myriad of other manipulations on CStrings. Listing 4-2 shows some sample CString manipulations:

Listing 4-2. Example CString manipulations

```
CString myString("This is a string that I'm going to manipulate!")
// Chop out Y (13) characters starting at X (22) and put it into
// newString.
CString newString = myString.Mid(22,5);
// yields: "I'm going to "
// Return the right ten characters
newString = myString.Right(10);
// yields: "This is a ".

// Nuke white space
myString.TrimRight();
// yields: "ThisisastringthatI'mgoingtomanipulate!"

// Upper case the string.
myString.MakeUpper();
// yields: "THISISASTRINGTHATI'MGOINGTOMANIPULATE!"

// Reverse the string
myString.MakeRevers();
// yields: "!ETALUPINAMOTGNIOGM'ITAHTGNIRTSASISIHT"
```

CString Reference Counting

With MFC 4.0, Microsoft introduced reference counting into class `CString`—a very welcome feature! Reference counting ensures that copies of the string data are made only when completely necessary. `CString` reference counting takes place automatically, but you can lock the string buffer with the `LockBuffer()` member function and unlock it via `UnlockBuffer()`. Reference counting reduces memory usage and increases performance, because fewer copies of the string are generated.

CString Internals

The `CString` features are pretty interesting, so let's see how class `CString` is implemented.

CString Files

The declaration for `CString` lives in `AFX.H`, and the implementation is spread between `AFX.INL` (inline file), `STRCORE.CPP`, and `STREX.CPP`.

The `CString` declaration is very long because of all the operators and member functions that it supports. Listing 4-3 contains an abbreviated version of the `CString`

declaration that highlights the implementation-specific and otherwise mostly undocumented parts of the class.

Listing 4-3. An abbreviated CString declaration, from AFX.H.

```
class CString
{
// NOTE: Lots of operators and member functions
// omitted for brevity..

// Implementation
public:
    ~CString();
    int GetAllocLength() const;

protected:
    LPTSTR m_pchData;      // pointer to ref counted string data
// implementation helpers
    CStringData* GetData() const;
    void Init();
    void AllocCopy(CString& dest, int nCopyLen, int nCopyIndex, int
                   nExtraLen) const;
    void AllocBuffer(int nLen);
    void AssignCopy(int nSrcLen, LPCTSTR lpszSrcData);
    void ConcatCopy(int nSrc1Len, LPCTSTR lpszSrc1Data, int nSrc2Len,
                    LPCTSTR lpszSrc2Data);
    void ConcatInPlace(int nSrcLen, LPCTSTR lpszSrcData);
    void FormatV(LPCTSTR lpszFormat, va_list argList);
    void CopyBeforeWrite();
    void AllocBeforeWrite(int nLen);
    void Release();
    static void Release(CStringData* pData);
    static int SafeStrlen(LPCTSTR lpsz);
};


```

m_pchData: The Key to Understanding CString

You can see in Listing 4-3 that CString has only one data member, LPTSTR m_pchData, which appears to be the pointer to the string that CString is encapsulating. But if CString is reference-counted, where is the count kept? How about the length of the string? Certainly it isn't calculated in every operation.

A closer look into the CString source code reveals some interesting undocumented facts about CString that answer these burning questions.

The CStringData Helper Structure

In AFX.H before the CString declaration, you will notice the CStringData helper structure declaration. This structure is shown in Listing 4-4.

Listing 4-4. The CStringData helper structure declaration

```
struct CStringData
{
    long nRefs;           // reference count
    int nDataLength;
    int nAllocLength;
    // TCHAR data[nAllocLength]

    TCHAR* data()
    { return (TCHAR*)(this+1); }
};
```

This helper structure seems to contain the CString information necessary for reference counting, but it is never used in AFX.H. Let's look at where `m_pchData` is allocated in `CString::AllocBuffer()` to figure out what's going on. `AllocBuffer()` gets called whenever `CString` needs to create space for a string. Listing 4-5 contains the psuedocode version of this undocumented `CString` routine.

Listing 4-5. The `CString::AllocBuffer()` implementation

```
void CString::AllocBuffer(int nLen)
{
    if (nLen == 0)
        Init();
    else{
        CStringData* pData =
            (CStringData*)new BYTE[sizeof(CStringData) +
            (nLen+1)*sizeof(TCHAR)];
        pData->nRefs = 1;
        pData->data()[nLen] = '\0';
        pData->nDataLength = nLen; pData->nAllocLength = nLen;
        m_pchData = pData->data();
    }
}
```

`AllocBuffer()` allocates a block of memory that is the size of a `CStringData` plus the size of the string requested. `AllocBuffer()` then initializes the `CStringData` members and finally sets the `m_pchData` pointer after the `CStringData` in the freshly allocated memory block. If no length was specified, `AllocBuffer()` calls `Init()`, which sets `m_pchData` to a special “empty” `CStringData` so that the functions can avoid checking for NULL all over the place. The “empty” `CStringData` is “prelocked” (its reference count is -1) so that they can also avoid manipulating its reference count. Figure 4-1 shows the `CStringData` allocation in `AllocBuffer()`.

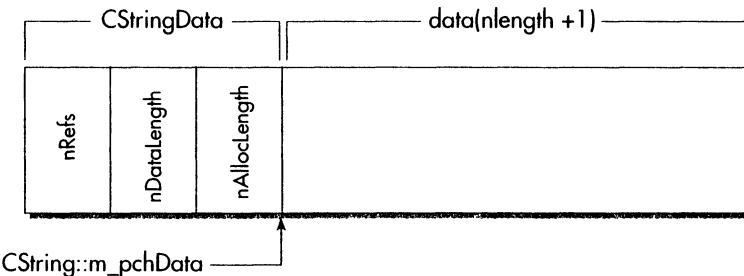


Figure 4-1. Memory allocated by AllocBuffer()

Aha! Class `CString` is hiding the information that it needs for reference counting before the data pointed to by `m_pchData`. Now, the million-dollar question is Why on earth would the MFC team do something wacky like this? Why not add some new members to `CString` and not risk playing with un-type-safe pointers like this?

The Scoop on `CStringData`

We asked Dean McCrory, our MFC insider, about the motivation for what seems like a really nasty “hack.” Here’s Dean’s reply:

The deal is that it is convenient that a `CString` object looks exactly like a `TCHAR*` in its binary form (that is, when you look at the bits that make up the two kinds of objects, they are the same). The reference count information isn’t kept in the `CString` itself because it is the data, not the `CString` itself, which is reference counted (a reference that exists in the referencing object doesn’t do a lot of good—the reference count must be in the referenced object). The two length members (`nDataLength` and `nAllocLength`) are in the `CStringData` object for a similar reason: they describe the actual data. They could have been carried around in each `CString`, but they’d be redundant for `CString` objects referencing the same data.

Microsoft could have also changed `m_pchData` to be a `CStringData` pointer and accessed the data that way in a more type-safe manner, but legacy MFC applications could potentially have code that derives a `CString` and accesses the `m_pchData` buffer pointer through the protected member, as in the example shown in Listing 4-6.

Listing 4-6. Example of code that could have been broken with MFC 4.0

```
//My string adds some new string functionality
class CMYString : public CString {
public:
    MyNeatFunction();
    //...
};
```

```
CMyString::MyNeatFunction()
{
    char buffer[256];
    strcpy(buffer, "I'm a WILD MFC HACKER! ");
    strcat(buffer,myString.m_pchData);
}
```

Changing the `m_pchData` member breaks any older MFC classes that access the `CString` buffer through the protected `m_pchData` member. The pains the MFC team takes to avoid breaking existing MFC are truly amazing!

According to Dean, there is a third reason for the hidden `CStringData`:

Another reason was a support issue. It was common for people to write this kind of code:

```
printf("The string %s was not found in file %s.",strSearch,
      strFileName);
```

This kind of code would blow up in previous versions of MFC because of the fact that `sizeof(CString)` was bigger than 4 and so `printf` would dereference something which wasn't a pointer. The corrected code is:

```
printf("The string %s was not found in file
      %s.",(LPCTSTR)strSearch,(LPCTSTR)strFileName);
```

Although obvious once you see what the problem is, it caused lots of questions to product support. When we did the `CString` reference counting we realized that we could, as a side effect of this work, allow the formerly broken code to work.

So the reasons for the hidden `CStringData` are threefold:

- To make `TCHAR` convenient and easy to use.
- To ensure backwards compatibility.
- To fix a common `CString` problem.

Reference Counting Revealed

Reference counting is by far the most interesting aspect of `CString`. Reference counting limits copying of string data to only those operations that really need to generate copies. (This usually includes operations that write or modify the string.) For example, consider the following code:

```
CString myString1("This is a string");
CString myString2(myString1);
CString myString3 = myString1;
```

As long as we never try to write to the string, we can conserve memory by having a single buffer that stores the “This is a string” and then point myString2 and myString3 to (or otherwise reference) myString1. Figure 4-2 shows this scenario.

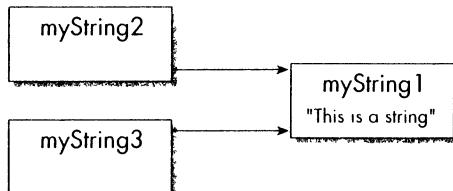


Figure 4-2. CString reference counting

Reference counting is a handy technique to have in your bag of MFC programming tricks because you can optimize your own MFC classes with a similar reference counting system. In fact, reference counting will become a very important topic when we cover OLE.

The `CStringData::nRefs` reference count is the key to the reference counting algorithm. The reference count is initialized to 0 and incremented for each reference made to the string. The reference count is decremented for each deleted reference, and if the reference count ever reaches 0 again, there are no more references, so the string can be deleted. If `CString` detects that a write is being made to a reference string, it automatically creates a copy of the referenced string and decrements the reference.

If you think about it, the data referenced by all of the `CStrings` can be considered a critical resource, just like most resources in the system-level operating system. What if two threads are running and try to increment the same reference at the same time? What about deleting?

MFC uses two Win32 APIs to take care of this problem: `InterlockedIncrement()` and `InterlockedDecrement()`. These nifty APIs automatically take care of all the operating system-level shared resource issues and guarantee that only one increment/decrement of the `nRefs` value will take place, no matter how many threads are trying to access the value simultaneously.

Let’s follow this example to figure out how the reference counting algorithm works. The numbers 1 through 5 represent the chronological execution of these lines of code:

1. `CString myString1("This is my string!");`
2. `CString myString2(myString1);`
3. `CString myString3; myString = myString2;`

```

4. // Write something!
5. myString3.MakeUpper();

```

With the MFC reference counting algorithm, the first three lines do not need copies of the data. In this case, myString1, myString2, and myString3 all reference the same string: "This is my string1!". In the fifth line, CString detects that a write is being made to the data, so CString creates new data. At this point there are two data objects: "This is my string1!", with two references (myString1,myString2), and "THIS IS MY STRING1!", with one reference (myString3).

Let's trace through the CString source code for this example and see firsthand how reference counting works.

Line 1 of the example is a call to the CString constructor that takes a char * as the argument. The pseudocode for this constructor, from STRCORE.CPP, is in Listing 4-7.

Listing 4-7. The CString constructor that gets called in line 1 of the example

```

CString::CString(LPCTSTR lpsz)
{
    Init();
    int nLen = SafeStrlen(lpsz);
    if (nLen != 0)
        AllocBuffer(nLen); memcpy(m_pchData, lpsz, nLen*sizeof(TCHAR));
}

```

The Init() member function initializes nRefs to -1 and zeroes out all of the other CStringData members. Next, the constructor determines the length of the argument and allocates a block of memory of that size by calling AllocBuffer(). (See Listing 4-5.) Finally, the CString constructor copies the argument into the newly allocated block of memory.

At this point, nRefs is set to 1 by AllocBuffer() for the sample CString variable, myString1.

Next, in line 2 the example calls the CString constructor that takes a CString reference as argument. Listing 4-8 contains the CString pseudocode for that constructor.

Listing 4-8. The CString constructor used in line 2 of the example

```

CString::CString(const CString& stringSrc)
{
    if (stringSrc.GetData()->nRefs >= 0){
        m_pchData = stringSrc.m_pchData;
        InterlockedIncrement(&GetData()->nRefs);
    }else{
        Init();
    }
}

```

```
        *this = stringSrc.m_pchData;  
    }  
}
```

If the number of references in the `stringSrc` argument is zero or greater, this constructor points its data pointer to `stringSrc`'s data. Note that the line

```
m_pchData = stringSrc.m_pchData;
```

also causes the string to share the complete CStringData information as well as the actual data. Therefore, the InterlockedIncrement() call increments the count referenced by both myString1 and myString2. The "else" clause is also important in understanding the effects of LockBuffer() and UnlockBuffer(). A CString that refers to data that is "locked" cannot be referenced by any other CString.

In addition, a “locked” CString cannot reference any other CString data other than its own. This is useful when you want to have particular CString instances that don’t share data with any other CStrings and do not use any other CString data—in other words, when you have particular CString instances that need to behave like non-reference-counted CStrings.

For example, consider the MFC ODBC classes. You must hand ODBC an address to "bind" to. It stuffs data into this address as you move about the database contents. The address cannot change or become deallocated. To make that happen, we call `CString::LockBuffer()` before binding to the address (`m_pchData`).

At this point the nRefs for both myString1 and myString2 are 2. Both CStrings point to the same data allocated by the first AllocBuffer() call.

The third line of the example first calls the default constructor for `myString3`, which initializes everything to `-1`, and it then calls the assignment operator. Listing 4-9 contains the pseudocode for the `CString` assignment operator.

Listing 4-9. The `CString` assignment operator called in line 3 of the example

```
const CString& CString::operator=(const CString& stringSrc)
{
    if (m_pchData != stringSrc.m_pchData) {
        if ((GetData()>nRefs < 0 && GetData() != afxDataNil) ||
            stringSrc.GetData()>nRefs < 0){
            // actual copy necessary since one of the strings is locked
            AssignCopy(stringSrc.GetData()>nDataLength,
                      stringSrc.m_pchData);
        }else{ // can just copy references around
            Release();
            m_pchData = stringSrc.m_pchData;
            InterlockedIncrement(&GetData()>nRefs);
        }
    }
}
```

```

    }
    return *this;
}
}

```

The example falls through to the “else” part of this operator, which first releases the buffer if necessary. In our example, this call to Release() does nothing because the reference count is not 0 or less. Next, the operator creates a reference to the myString1 data and increments the nRefs reference count to 3.

The “if” part of the assignment operator creates a copy of the buffer if the string is locked (nRefs = -1).

At this point in the example, on line 4, after myString3 is constructed and before MakeUpper() is called, all three CString objects—myString1, myString2, and myString3—point to the original data allocated with AllocBuffer().

In line 5, the reference count example calls the CString::MakeUpper() member function, which causes a write to the shared buffer and thus prompts CString to copy the buffer so that all three strings don’t end up with the modified all-uppercase string. Listing 4-10 shows the pseudocode for MakeUpper().

Listing 4-10. The MakeUpper() member function called on line 5 of the example

```

void CString::MakeUpper()
{
    CopyBeforeWrite();
    ::CharUpper(m_pchData);
}

```

The CString::CopyBeforeWrite() member function does the work of copying the buffer and updating the reference counts, so let’s look at that function next.

The pseudocode for CopyBeforeWrite() is in Listing 4-11.

Listing 4-11. The CopyBeforeWrite() member function, which copies the string data before a write operation

```

void CString::CopyBeforeWrite()
{
    if (GetData()->nRefs > 1){
        CStringData* pData = GetData();
        Release();
        AllocBuffer(pData->nDataLength);
        memcpy(m_pchData, pData->data(),
               (pData->nDataLength+1)*sizeof(TCHAR));
    }
}

```

First, CopyBeforeWrite() verifies that the reference count is greater than 1 and that a copy is really needed. Next, CopyBeforeWrite() releases the reference via the

Release() member function and allocates a fresh buffer for the data, with nRefs = 1. Finally, CopyBeforeWrite() copies the data from the previously referenced string to the fresh, newly allocated string data area.

That concludes the reference-counting example. Once the example code runs, there will be two sets of string data: the one shared by myString1 and myString2, and the private copy of the uppercased string referenced by myString3.

Before we move onto other CString features, let's take a look at the Release() member function to see exactly how it removes a reference. The pseudocode for CString::Release() is in Listing 4-12.

Listing 4-12. CString::Release(), the member function that removes a CString reference

```
void CString::Release()
{
    if (GetData() != afxDataNil){
        ASSERT(GetData()->nRefs != 0);
        if (InterlockedDecrement(&GetData()->nRefs) <= 0)
            delete[] (BYTE*)GetData();
        Init();
    }
}
```

After checking that (1) the string is not NULL and (2) there are indeed references to be released, the Release() member function calls InterlockedDecrement(). If the reference count after the decrement is 0 or less, then there are no more references to the data (not even the “this” CString) and the data should be freed. In this case, Release() calls delete on the data. The CString destructor makes this same check so that buffers are not erroneously destroyed when a CString is deleted.

Wrapping Up Reference Counting

The CString reference count algorithm is pretty easy to follow once you know about some of the tricks MFC uses to hide the reference counting. As an exercise, you should browse through STRCORE.CPP and note which functions call CopyBeforeWrite(). Also, see if you can figure out why the operator += does not call CopyBeforeWrite().

Other Interesting CString Internals

The majority of CString member functions rely on one of the _t internationalized string functions. For example, the pseudocode for CString::Find() is in Listing 4-13.

Listing 4-13. CString::Find()—an example of how CString calls the _t string routines

```
int CString::Find(TCHAR ch) const
{
```

```

// find first single character
LPTSTR lpsz = _tcschr(m_pchData, ch);

// return -1 if not found and index otherwise
return (lpsz == NULL) ? -1 : (int)(lpsz - m_pchData);
}

```

The Find() member function simply passes the argument and internal string pointer to _tcschr(), which gets mapped to the familiar strchr() for single-byte strings. After calling _tcschr(), Find() changes the result of the call to the documented return value.

Moving On Along . . .

Now that we've dissected the most complex of the value type classes, let's look at a couple of the interesting internals of some other value type classes.

Other Simple Value Types

Of course, CString is but one of several simple value types in MFC. Unlike CString, the other simple value types are almost completely inlined and don't add nearly as many features to the encapsulated type.

The following table lists the non-CString simple value types, the structure they encapsulate, and the source files where they are implemented:

<i>Value Type</i>	<i>Structure</i>	<i>Source File</i>
CPoint	POINT (struct tagPOINT)	afxwin1.inl
CRect	RECT (struct tagRECT)	afxwin1.inl, wingdix.cpp
CSIZE	SIZE (struct tagSIZE)	afxwin1.inl
CTime	time_t operations	afx.inl, timecore.cpp
CTimeSpan	time_t math	afx.inl, timecore.cpp

Let's take a look at the CRect class, because its implementation is very similar to all the others.

Class CRect

CRect is declared in AFXWIN.H as a derivative of tagRECT:

```
class CRect : public tagRECT
```

The tagRECT is a standard Windows structure that lives in WINDEF.H and is declared there as

```

typedef struct tagRECT
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;

```

By deriving CRect from tagRECT, MFC maintains layout compatibility between CRect and tagRECT. That means you can use a regular tagRECT wherever CRect is listed in a function's parameter list. In addition, MFC adds functionality to tagRECT without having to duplicate the left/top/right/bottom data members.

All of the CRect member functions manipulate these four data members in various ways. For example, the DeflateRect() member function shrinks the current rectangle based on the sizes found in an argument rectangle. The pseudocode for DeflectRect() from WINGDIX.CPP is in Listing 4-14.

Listing 4-14. CRect::DeflateRect(), a typical simple value type member function

```

void CRect::DeflateRect(LPCRECT lpRect)
{
    left += lpRect->left;
    top += lpRect->top;
    right -= lpRect->right;
    bottom -= lpRect->bottom;
}

```

This convenience member function, like most of the other simple value type member functions, keeps the programmer from having always to access all four data members.

If you've seen one simple value type class, you've seen them all. So let's move on and see how MFC implements the next category of utility classes, the collections.

MFC Collection Classes

MFC provides three different kinds (or shapes) of canned abstract data structures that store (or collect) application data for you. For each shape of collection, MFC supplies several nontemplate classes that hold types such as WORD, int, BYTE, CString, and CObject. MFC also provides template versions of each shape, so that you can create a type-safe collection for your own classes.

MFC Collection Shapes

The three shapes supported by MFC are arrays, lists, and maps. Arrays are just like C++ arrays, except they handle all of the memory allocation for you. MFC lists are doubly linked lists. Maps, sometimes called dictionaries, store pairs of values so that you can quickly look up a key and obtain its value.

Table 4-1 lists the names of all the MFC collection classes, their shape, and the type(s) that they store.

Table 4-1. The MFC collections

Name	Shape	Type1	Type2
CByteArray	Array	BYTE	
CDWordArray	Array	DWORD	
CUIntArray	Array	unsigned int	
CObArray	Array	CObject	
CStringArray	Array	CString	
CWordArray	Array	WORD	
CObList	List	CObject*	
CPtrList	List	void*	
CStringList	List	CString	
CMapPtrToPtr	Map	void*	void*
CMapStringToOb	Map	CString	CObject*
CMapStringToPtr	Map	CString	void*
CMapStringToString	Map	CString	CString
CMapWordToOb	Map	WORD	CObject*
CMapWordToPtr	Map	WORD	void*

In this section we'll investigate each MFC collection shape, starting with arrays, moving on to lists, and finally covering maps. Besides being a good refresher in abstract data types, each MFC collection has some interesting techniques that can be applied to any C++ class.

MFC Array Collections

Let's start by reviewing an example illustrating how to use the MFC array collections. Then we'll examine how an array implemented.

Array Review

Using an MFC array is fairly straightforward. Just initialize the array with the SetSize() member function and then add items to the array with the SetAt() member

function. To retrieve values in the array, use the `GetAt()` method or the familiar `[]` operator. Use `DeleteAt()` to remove an item from an array.

The example in Listing 4-15 places twice the index into each “slot” in a 100-element unsigned int array.

Listing 4-15. An example use of an array

```
void ArrayExample()
{
    CUIntArray myIntArray;
    //Initialize size
    myIntArray.SetSize(100);

    //Fill it up with 2 times index.
    for (int i = 0; i < 100;i++)
        myIntArray.SetAt(i,2*i);
    // Now access item 50
    UINT bogus = myIntArray[50];
    //Alternative access
    bogus = myIntArray.GetAt(50);
    //Delete item 50
    myIntArray.DeleteAt(50);
}
```

MFC Array Internals

The declarations for the nontemplate array classes are in the AFXCOLL.H header file. The implementations are in the source files: ARRAY_X.CPP, where X is the type of array. For example, the `CStringArray` is located in `ARRAY_S.CPP`. All of the MFC template collection classes live in the AFXTEMPL.H header file.

The various `CArray` flavors have the exact same implementation, but with different data types. Let’s look at the `CWordArray` class (an array of words) and use it as an example of how MFC implements all of the array collections.

Listing 4-16 contains an abbreviated version of the `CWordArray` declaration, with the documented constructors/attributes and operations omitted so that we can focus on the `// Implementation` section.

Listing 4-16. The declaration of `CWordArray`, from AFXCOLL.H (with documented members omitted)

```
class CWordArray : public CObject
{

DECLARE_SERIAL(CWordArray) public:
// Construction **omitted
// Attributes **omitted
```

```

// Operations **omitted
// ...

// Implementation -The undocumented parts!!
protected:
    WORD* m_pData;      // the actual array of data
    int m_nSize;        // # of elements (upperBound - 1)
    int m_nMaxSize;     // max allocated
    int m_nGrowBy;      // grow amount

public:
    ~CWordArray();
    void Serialize(CArchive&);
};

}

```

All of the CArray data members are undocumented, so here's a quick description of what each is used for:

- **m_pData**—This data pointer points to the actual array data. The type of this member varies in each array. For example, it is a **CString** pointer in a **CStringArray** collection.
- **m_nSize**—The current size of the array, set by **SetSize()**.
- **m_nMaxSize**—The internal array grows in chunks larger than WORD, so there are “extra” elements that are allocated but not initialized. **m_nMaxSize** tracks the “physical” size of the array, where **m_nSize** tracks the “logical” size, or the size visible to the developer.
- **m_nGrowBy**—The number of elements to allocate in each allocated chunk of memory. MFC provides a best-guess grow-by size that minimizes memory fragmentation.

CWordArray Implementation

You can find the implementation for **CWordArray** in the **ARRAY_W.CPP** (**w = WORD**). The most interesting facet of MFC arrays is how they manage memory.

One note before starting: Every **CArray** operation that accesses a position in the array checks to see if memory is allocated for the position at that index. If not, **CArray** allocates more memory through the **SetSize()** member function.

How Arrays Grow: The **SetSize()** Member Function

SetSize() is the key to understanding the **CArray** memory handling, so let's look at it now, as shown in Listing 4-17.

Listing 4-17. The pseudocode implementation of CWordArray::SetSize()

```

void CWordArray::SetSize(int nNewSize, int nGrowBy)
{
    if (nGrowBy != -1)
        m_nGrowBy = nGrowBy; // set new size
    if (nNewSize == 0){ //shrink to nothing
        delete[] (BYTE*)m_pData;
        m_pData = NULL;
        m_nSize = m_nMaxSize = 0;
    } else if (m_pData == NULL){ //create on with exact size
        m_pData = (WORD*) new BYTE[nNewSize * sizeof(WORD)];
        // zero fill m_nSize = m_nMaxSize = nNewSize;
        memset(m_pData, 0, nNewSize * sizeof(WORD));
    } else if (nNewSize <= m_nMaxSize){ //use an already allocated block
                                         //of memory
        if (nNewSize > m_nSize)
            memset(&m_pData[m_nSize], 0, (nNewSize-m_nSize) * sizeof(WORD));
        m_nSize = nNewSize;
    }
    else{ // Grow array
        int nGrowBy = m_nGrowBy;
        if (nGrowBy == 0)
            nGrowBy = min(1024, max(4, m_nSize / 8));
        int nNewMax;
        if (nNewSize < m_nMaxSize + nGrowBy)
            nNewMax = m_nMaxSize + nGrowBy; // granularity
        else
            nNewMax = nNewSize; // no slush
        WORD* pNewData = (WORD*) new BYTE[nNewMax * sizeof(WORD)];
        // copy new data from old
        memcpy(pNewData, m_pData, m_nSize * sizeof(WORD));
        // construct remaining elements
        memset(&pNewData[m_nSize], 0, (nNewSize-m_nSize) * sizeof(WORD));
        // get rid of old stuff (note: no destructors called)
        delete[] (BYTE*)m_pData;
        m_pData = pNewData;
        m_nSize = nNewSize;
        m_nMaxSize = nNewMax;
    }
}

```

The SetSize() member function breaks down into five sections.

1. The initialization section—SetSize() starts by setting the m_nGrowBy value if it is different from the -1 default.

2. The shrink-to-nothing section—The “if (*nNewSize == 0*)” block is executed if the user has called SetSize(0). The *m_pData* is freed and the member variables are all set to zero.
3. The first allocation section—The “else if (*m_pData == NULL*)” block is executed if the previous size was zero, which is usually the case for the first SetSize() call. This section allocates a block of memory that is exactly the size requested and initializes it to zero using memset(). The code block then sets *m_nSize* and *m_nMaxSize* to the new size. For example, if you call SetSize(0) and then SetSize(100), this is the block that will get executed.
4. The allocate-from-extra-space section—The “else if (*nNewSize <= m_nMaxSize*)” condition checks to see if the requested new size fits inside the “physical” size of the array. If it does, it initializes those elements to zero using memset() and then updates *m_nSize* to reflect the new logical size.
5. The grow-the-array section—The last block, which starts with the “else,” is called if an array already exists but needs to be grown. This section does most of the work.

First, if an *nGrowBy* was not specified, this “else” block sets *nGrowBy* using the result of the following calculation:

```
min(1024, max(4,m_nSize/8);
```

This calculation helps avoid heap fragmentation by keeping memory growth greater than 3. The 1024 check is somewhat arbitrary in that a large grow-by size can tend to allocate more memory than you end up using. Microsoft felt that a maximum of 1023 extra unused elements was a good limit by default. It is best if you look at your own data and either set the size up front or set the grow-by value to something appropriate.

The next part of this section sets *nNewMax* to the sum of the requested size and the grow-by size. Then the new data is allocated, and the old data is copied from the old *m_pData* into the newly allocated area using memcpy. The elements that fall into the requested size are initialized to zero and the old data is erased. The *m_pData* data member is updated to point to the new memory. Finally, the *m_nSize* and *m_nMaxSize* member variables are updated.

The last block of code is fairly expensive, because it has to allocate, copy, and deallocate potentially large chunks of data. The extra grow-by memory tagged onto each allocation helps cushion the effects of this block. As long as the array grows inside the grow-by amount, another allocate/copy/free sequence won’t have to be called.

Adding an Element to an Array: SetAt() and InsertAt()

Now that we've seen how arrays are allocated, let's see how MFC puts items into them. This happens in the implementation of SetAt() and InsertAt(). SetAt() lets you put an element in an array, and InsertAt() inserts an item into the array.

SetAt()'s implementation is so short that it's inlined in the AFXCOLL.INL file. Here's the one-line implementation of SetAt():

```
m_pData[nIndex] = newElement;
```

SetAt() is accessing the internal array as you would a normal C++ array.

InsertAt() is a little longer and much more expensive, because it has to shift literally all of the indexes equal to and greater than the destination by one. Listing 4-18 contains the pseudocode for InsertAt().

Listing 4-18. The implementation of CWordArray::InsertAt()

```
void CWordArray::InsertAt(int nIndex, WORD newElement, int nCount)
{
    if (nIndex >= m_nSize) // adding after the end of the array
        SetSize(nIndex + nCount); // grow so nIndex is valid
    else { // inserting in the middle of the array
        int nOldSize = m_nSize;
        SetSize(m_nSize + nCount); // grow it to new size
        // shift old data up to fill gap
        memmove(&m_pData[nIndex+nCount], &m_pData[nIndex],
                (nOldSize-nIndex) * sizeof(WORD));
        // re-init slots we copied from
        memset(&m_pData[nIndex], 0, nCount * sizeof(WORD));
    }
    // insert new value in the gap
    while (nCount--)
        m_pData[nIndex++] = newElement;
}
```

If the user is inserting the element at the end of the array, InsertAt() grows the list to fit the new element via SetSize() and then inserts the new value.

If the user is inserting the element into the middle of the array, InsertAt() grows the array to the new size and then uses memmove to shift the array data up one. Finally, the new value is inserted into the gap. Keeping this internal detail in mind will help you avoid overusing the expensive InsertAt() method.

Retrieving Data from the Array through GetAt() and Operator []

Before moving onto MFC lists, let's look at how MFC supports accessing the array through the GetAt() and operator [] member functions. Both of these member functions

are located in the inline file AFXCOLL.INL. Listing 4-19 contains the pseudocode for both.

Listing 4-19. The MFC array-retrieval members

```
WORD CWordArray::operator[](int nIndex) const
{
    return GetAt(nIndex);
}

WORD CWordArray::GetAt(int nIndex) const
{
    return m_pData[nIndex];
}
```

As you can see from this code, operator [] actually calls GetAt(), which is a normal way to access the m_pData member. By making all access to the array go through GetAt(), the MFC team can change the implementation later by changing only the code in GetAt(). This is a great data abstraction technique to use in your own C++ classes.

To summarize, the MFC arrays for the most part encapsulate C++ language-type arrays. They do add some interesting memory handling, which we covered in the SetAt() member function. Because of this memory handling, some member functions should be avoided like the plague, such as InsertAt(). You can also fine-tune the memory-handling scheme by providing your own grow-by factor that is best suited to your use of the array.

The next type of MFC collection that we'll cover is the infamous linked list.

Lists

This section reviews the use of MFC list collections and then dissects the CStringList to reveal a completely undocumented class, the (oddly named) CPlex.

The MFC list collections are doubly linked list and support vanilla list operations such as these:

- AddHead()—Adds an element to the front of the list.
- Find()—Searches for an element in the list.
- GetCount()—Retrieves the number of elements in the list.
- GetHead()—Gets the first element (head) in the list.
- GetNext()—Gets the next element given an element's POSITION.
- GetTail ()—Gets the last element in the list.

- IsEmpty()—Determines if the list is empty.
- RemoveAll()—Removes all elements from the list.
- RemoveHead()—Removes the first element from the list.
- RemoveTail()—Removes the last element from the list.
- InsertAfter()—Inserts an element after the given POSITION.
- InsertBefore()—Inserts an element before the given POSITION.

Items of a list can be accessed through a POSITION iterator. There are member functions to retrieve the POSITION of a certain element and also member functions to retrieve an element based on a certain POSITION. For example, here is some code that iterates through a CStringList:

```
// ... m_strList is the string list.
POSITION pos = m_strList.GetHeadPosition();      while (pos != NULL)
CString tmpString = m_strList.GetNext(pos);
```

MFC List Internals

Let's use the CStringList, a list of strings, as the focus of our list investigation. The other lists are very similar, except instead of CStrings they contain CObject pointers or void pointers. The declaration for CStringList is located in the AFXCOLL.H header file. The implementation lives in the LIST_S.CPP source file.

The CStringList Declaration

The declaration for CStringList appears in Listing 4-20. As with CWordArray, some of the declarations of the documented member functions have been omitted so that we can focus on the undocumented details of the class.

Listing 4-20. The CStringList declaration, from AFXCOLL.H

```
class CStringList : public CObject
{
DECLARE_SERIAL(CStringList)
protected:
    struct CNode {
        CNode* pNext;
        CNode* pPrev;
        CString data;
    };
public:
// Construction **omitted.
// Attributes (head and tail) **omitted
// Operations **omitted
```

```

// iteration ** omitted
// Implementation
protected:
    CNode* m_pNodeHead;
    CNode* m_pNodeTail;
    int m_nCount;
    CNode* m_pNodeFree;
    struct CPlex* m_pBlocks;
    int m_nBlockSize;

    CNode* NewNode(CNode*, CNode*);
    void FreeNode(CNode*);

public:
    ~CStringList();
    void Serialize(CArchive&);

};


```

This class contains an embedded declaration of a CNode structure, which is the actual element, or node, of the linked list. As you would expect with a doubly linked list, CNode is composed of the CString data field and a pair of pNext/pPrev pointers.

In the // Implementation section of CStringList, we find some typical list items:

- m_pNodeHead—A CNode pointer to the first element of the list (the head).
- m_pNodeTail—A CNode pointer to the last element in the list (the tail).
- m_nCount—The number of elements in the list.

There are three undocumented data members in CStringList: m_pNodeFree, m_pBlocks, and m_nBlockSize. There are also two undocumented member functions: NewNode() and FreeNode().

To understand what these members do, let's first take a look at the new helper structure that m_pBlocks is a pointer to, a CPlex.

Cplex: The Undocumented Collection Memory Helper

The declaration for the CPlex structure is in the AFXPLEX_.H private header file and is shown in Listing 4-21. The CPlex implementation is located in the PLEX.CPP MFC source file.

Listing 4-21. The undocumented CPlex structure used by MFC collection memory management, from AFXPLEX_.H

```

struct CPlex
{
    CPlex* pNext;
    UINT nMax;
};


```

```

    UINT nCur;
    void* data() { return this+1; }

    static CPlex* PASCAL Create(CPlex*& head, UINT nMax, UINT cbElement);
    void FreeDataChain();
};

}

```

From this declaration it appears that CPlex is yet another node declaration. There's a pNext pointer, so you can tell that this is a singly linked node list. The data is the memory allocated directly after the class (this+1). To understand how this new node type is used, let's look at the CStringList::NewNode() member function. Don't worry, we'll return to the CPlex members nMax, nCur, Create(), and FreeDataChain() after we've uncovered what CPlex is used for in CStringList::NewNode().

Following the CPlex Trail: CStringList::NewNode()

When you look through LIST_S.CPP, it's pretty obvious that NewNode() is the critical area for the CStringList memory management. All of the Add() and Insert() member functions call NewNode() first to allocate a node before it's added to the list. NewNode() is also the only member that touches the m_pBlocks CPlex pointer that we are investigating. The pseudo-code for CStringList::NewNode() is in Listing 4-22.

Listing 4-22. The source of all list allocations: the CStringList::NewNode() implementation, from LIST_S.CPP

```

CStringList::CNode*
CStringList::NewNode(CStringList::CNode* pPrev, CStringList::CNode* pNext)
{
    if (m_pNodeFree == NULL){// add another block
        CPlex* pNewBlock = CPlex::Create(m_pBlocks, m_nBlockSize,
                                         sizeof(CNode));
        // chain them into free list
        CNode* pNode = (CNode*) pNewBlock->data();
        // free in reverse order to make it easier to debug
        pNode += m_nBlockSize - 1;
        for (int i = m_nBlockSize-1; i >= 0; i--, pNode--)
            pNode->pNext = m_pNodeFree; m_pNodeFree = pNode;

    }

    CStringList::CNode* pNode = m_pNodeFree;
    m_pNodeFree = m_pNodeFree->pNext;
    pNode->pPrev = pPrev;
    pNode->pNext = pNext;
    m_nCount++;
    ASSERT(m_nCount > 0); // make sure we don't overflow
}

```

```

ConstructElement(&pNode->data);
return pNode;
}
}

```

NewNode() first checks if there are any free CNodes pointed to by m_pNodeFree. If there are not free CNodes, NewNode() calls the CPlex::Create() member function, with m_nBlockSize as the block size and sizeof(CNode) as the size of the element. CPlex::Create() returns a CPlex pointer that contains memory for m_nBlockSize CNode list elements.

To understand the NewNode() code after the call to CPlex::Create(), it helps to know that the MFC list maintains a singly linked list of free CNode elements. The m_pNodeFree points to the head of this list, and the elements of the list are chained together using the CNode->pNext pointer. It's pretty interesting (and sometimes confusing) that MFC uses the pNext pointer in one of two contexts: when the linked list is in use, and when MFC is maintaining the free list.

The code after Create() iterates through the new block of memory and puts the CNodes there onto the free list from the end of the memory to the front.

After NewNode() finishes allocating and placing new CNodes onto the free list, it removes a fresh CNode from the free list and updates the free list to reflect that it has taken a CNode. Next, NewNode() updates the CNode pPrev and pNext pointers to point to the pPrev and pNext arguments. Finally, NewNode() increments the internal node count and calls ConstructElement, which zeroes out the data portion of the CNode.

Note: The CString collections use a common set of undocumented utility functions that can be found in the SRC\ELEMENTS.H header file. Though we aren't going to cover them in detail here, *these utility functions warrant examination* because some may be useful in your own CString collections.

Back to CPlex

We've seen from NewNode() that a CPlex is basically a block of memory that gets chopped into CNodes. Let's look at the CPlex Create() function to see how the unknown CPlex members nMax, nCur, and pNext are used. Listing 4-23 contains the pseudocode for CPlex::Create(), which can be found in PLEX.CPP.

Listing 4-23. The CPlex::Create() implementation, from PLEX.CPP

```

CPlex* PASCAL CPlex::Create(CPlex*& pHead, UINT nMax, UINT cbElement)
{
    CPlex* p = (CPlex*) new BYTE[sizeof(CPlex) + nMax * cbElement];
    // may throw exception
    p->nMax = nMax;
    p->nCur = 0;
    p->pNext = pHead;
}

```

```

    pHead = p; // change head (adds in reverse order for simplicity)
    return p;
}

```

Looking at this function, we can see that pNext is used for yet another linked list! This linked list stores the various CPlex blocks of allocated memory in the reverse order.

Unraveling the Mysteries of CPlex!

The real mystery is why the CPlex structure has the nMax and nCur members. These members are set in Create() but never used anywhere in the MFC source code. We asked our MFC insider, Dean McCrory, about these mysterious CPlex members. He replied: "They appear to be unused. The only explanation that I can offer is that the CPlex structure dates back to the 'old AFX,' the effort before MFC to create a framework (by the same group of people)."

After we revealed these unused members to Dean, he proceeded to rip them out of MFC! So in MFC 4.0, you will notice in AFX.H:

```
#define CPlex CPlexNew
```

And notice the following comment at the top of PLEX.CPP:

```
// CPlexNew - CPlex without nCur and nMax members
```

Dean uses this trick of redefining the name of the class so that old MFC applications do not break.

Why get rid of these unused members? Well, remember that CPlex is the key memory mechanism for both lists and maps, so each CPlex structure allocated wastes these two integers. And the wasted memory is compounded because MFC uses lists and maps internally. Your MFC applications could have thousands (or more!) CPlex blocks that all waste the two integers.

Think of all the cumulative memory saved from not having to allocate those two integers in every CPlex! MFC internals: changing the future of MFC programming for the better.

Free, Free, Set Them Free!

As you look through the LIST_S.CPP file, you'll notice that elements are almost never truly "freed" from the list. If an element is deleted, it is simply added to the free list, but not truly freed. The only place where memory is really freed is in CStringList::RemoveAll(). Let's look at that member function to see how the elements are freed. The pseudocode is in Listing 4-24.

Listing 4-24. CStringList::RemoveAll(): How MFC frees the CNodes from a list

```
void CStringList::RemoveAll()
{
    CNode* pNode;
    for (pNode = m_pNodeHead; pNode != NULL; pNode = pNode->pNext)
        DestructElement(&pNode->data);
    m_nCount = 0;
    m_pNodeHead = m_pNodeTail = m_pNodeFree = NULL;
    m_pBlocks->FreeDataChain();
    m_pBlocks = NULL;
}
```

First, RemoveAll() iterates through the list of elements and frees the elements themselves. (Note: DestructElement is another one of those handy utility functions from ELEMENTS.H. Next, RemoveAll() zeroes out the count and all of the pointers. Finally, RemoveAll() calls the CPlex::FreeDataChain() member function and then nulls out the m_pBlocks CPlex list of allocated memory pointers.

It's still not obvious where the memory is getting freed in RemoveAll(), so let's look at the FreeDataChain() member function. Listing 4-25, from PLEX.CPP, contains the short CPlex::FreeDataChain() function.

Listing 4-25. CPlex::FreeDataChain(): Where all CPlex allocated memory blocks get freed

```
void CPlex::FreeDataChain()
{
    CPlex* p = this;
    while (p != NULL){
        BYTE* bytes = (BYTE*) p;
        CPlex* pNext = p->pNext;
        delete[] bytes;
        p = pNext;
    }
}
```

FreeDataChain() walks the CPlex allocated memory block list and deletes each CPlex node in that list. Note that FreeDataChange() always grabs the next pointer *before* deleting the node, so that it can continue its destructive trip down the list without acting like Wile E. Coyote and sawing off the tree limb underneath himself.

CList Memory Management Summary

One key point to remember about CList memory management is that MFC maintains three lists for each CList:

1. The actual in-use list, a doubly linked list of CNodes, whose head is pointed to by m_pNodeHead and whose tail is pointed to by m_pNodeTail.
2. A list of free nodes. The head of this list is pointed to by m_pNodeFree.
3. A list of all allocated memory blocks, or CPlexes. The head of this list is pointed to by m_pBlocks.

Another key point about MFC lists is that when you delete elements, memory is not really freed but is reused via the free list. Your lists can grow very large if you do lots of additions and deletions. The RemoveAll() member function does free all the memory for you through the nifty but destructive CPlex::FreeDataChain() member function.

Now that you understand how MFC implements memory management for its lists, you can use the same approach in any of your custom data structures that are “node” based. By using this approach, you do not have to allocate/deallocate very fine grain pieces of memory (for example, a CNode versus a CPlex), which is very expensive.

Inside MFC List Iteration

In the review of how to use MFC lists, you might have noticed a kind of vague type, POSITION. The definition of POSITION lives in AFX.H:

```
// abstract iteration position
struct __POSITION { int unused; };
typedef __POSITION* POSITION;
```

The POSITION is similar to a void pointer; it actually contains a CNode pointer. However, this fact is hidden from the class user because that implementation fact could change. So now you know!

Another Unsolved Mystery!?

Why is POSITION a typedefed pointer to a struct that contains an unused int? It's not really clear. In MFC versions before 4.0, this was a true void pointer.

We put this unsolved mystery to our MFC insider, Dean McCrory. His response:

Type safety. We found it was pretty common where people would pass POSITIONs in the wrong param, like when enumerating a map or list of void*. Because the types of POSITION and the type of the collection were the same, you could accidentally pass a POSITION where one was not expected or pass a void* where a POSITION was expected. Using the technique above (which is similar to the STRICT stuff that was added to WINDOWS.H in Win31) allows better type checking by the compiler.

To prove that a POSITION is a pointer to a CNode, here's the implementation of GetHeadPosition() from the AFXCOLL.INL inline file:

```
POSITION CPtrList::GetHeadPosition() const
{
    return (POSITION) m_pNodeHead;
}
```

The CStringList::GetNext() member function is also found in the AFXCOLL.INL file:

```
CString& CStringList::GetNext(POSITION& rPosition)
{
    CNode* pNode = (CNode*) rPosition;
    rPosition = (POSITION) pNode->pNext;
    return pNode->data;
}
```

For More Information

The other CStringList member functions that insert, delete, and find elements are pretty straightforward linked-list manipulations. Now that you've seen the way POSITIONs work, you can check out these routines. You'll notice how crucial the NewNode() and FreeNode() functions are once you've looked at a couple of them!

MFC Map Collections

In this section we'll see how MFC implements its map collections. But first we'll review the basics of using maps.

Maps, sometimes called dictionaries, store pairs of information; each pair has a *key* and a *value*. The key is used to look up the value. MFC maps use a hash table scheme for storing the keys; therefore, maps provide much faster lookup than lists, which must be searched.

MFC calls the key/value pairs an *association*. Listing 4-26 shows some example map code that uses the CMapStringToString map collection.

Listing 4-26. An example of CMapStringToString usage

```
//...
CMapStringToString myStringMap;

//Let's create a spanish dictionary!
myStringMap.SetAt("one", "uno");
myStringMap.SetAt("two", "dos");
myStringMap.SetAt("three", "tres");
```

```

//... sometime later, the user wants to convert 'one' to spanish..
CString strAnswer;
if (myStringMap.LookUp("one", answer))
    // got it, should be uno.
else
    // oops, wasn't in there?!

// Iterate through the map
POSITION pos = GetStartPosition();
while (pos != NULL){
    CString key, value;
    GetNextAssoc(pos, key,value);
    //Down here, key and value have been
    //Updated to hold the next association
}

```

In addition to the operations highlighted by the CMap example, there are member functions for removing associations, getting the number of associations, and performing other normal collection operations.

The CMap Implementation

Let's look at how MFC implements one of the maps that has keys and values of different type, such as CMapWordToPtr.

The declaration for CMapWordToPtr can be found in AFXCOLL.H, and the implementation lives in MAP_WP.CPP. The CMap source files are names using this scheme: MAP_XY.CPP, where X is the key of the map and Y is the value of the map. In our case, key = WORD and value = PTR (void*).

Listing 4-27 contains an abbreviated CMapWordToPtr declaration, from AFXCOLL.H:

Listing 4-27. The CMapWordToPtr declaration, from AFXCOLL.H

```

class CMapWordToPtr : public CObject
{
DECLARE_DYNAMIC(CMapWordToPtr) protected:
// Association
    struct CAssoc{
        CAssoc* pNext;
        UINT nHashValue; // needed for efficient iteration WORD key;
        void* value;
    };
public:
// Construction **omitted
// Attributes **omitted
// Operations **some omitted

```

```

// advanced features for derived classes
UINT GetHashTableSize() const;
void InitHashTable(UINT hashSize, BOOL bAllocNow = TRUE);

UINT HashKey(WORD key) const;
// Implementation
protected:
    CAssoc** m_pHashTable;
    UINT m_nHashTableSize;
    int m_nCount;
    CAssoc* m_pFreeList;
    struct CPlex* m_pBlocks;
    int m_nBlockSize;

    CAssoc* NewAssoc();
    void FreeAssoc(CAssoc* );
    CAssoc* GetAssocAt(WORD, UINT&) const;

public:
    ~CMapWordToPtr();
};

```

If you check the MFC documentation for any of the `CMap` classes, you will notice that there are lots of member functions and data from Listing 4-27 that are completely undocumented! That means there's more to discover, so let's get cracking.

First, notice that like `CList::CNode`, all of the map collections have an embedded `CAssoc` structure declaration that describes the element of the map. The `CAssoc` structure holds the key and value. `CAssoc` also appears to have some kind of linked list maintained through the `pNext` pointer. There's also an `nHashValue` that we'll investigate later.

Look Familiar?

Some of the undocumented member functions should look familiar after our coverage of the MFC list collections. Maps, like lists, use the same `CPlex` memory helper structure and algorithm for memory handling. With this in mind, we can already guess what these `CMapWordToPtr` members do:

- `m_nCount`—The number of elements in the map.
- `m_pBlocks`—A pointer to the head of a list of the allocated `CPlex` blocks.
- `m_pFreeList`—A pointer to a list of free `CAssocs` allocated from `CPlexes`.
- `m_nBlockSize`—The size of a `CPlex` block to be allocated.

Because we've already covered class `CPlex` in gory detail, let's jump straight to the other interesting undocumented facet of maps, the hashing.

MFC Map Hashing

If you are unfamiliar with hashing, here's a quick review. Hashing is used to provide speedy lookups. You run your key through a hash function, which generates the hash table index into which the data is stored. To retrieve the value, you run your key through the same hash function to get the index and—viola!—you have your data without having to search for it.

The MFC map hashing is a classic example of hashing. First, we'll look at the hash table and how and when it is allocated. Next, we'll look at how keys are hashed into the table. Finally, we'll look at how values are retrieved from the hash table.

All of the MFC maps allocate the internal hash table in the `InitHashTable()` call. Listing 4-28 contains the pseudocode for the `InitHashTable()` member function from `MAP_WP.CPP`.

Listing 4-28. Where the map hash table is initialized—`InitHashTable()`

```
void CMapWordToPtr::InitHashTable(UINT nHashSize, BOOL bAllocNow)
{
    if (m_pHashTable != NULL) {
        // free hash table
        delete[] m_pHashTable;
        m_pHashTable = NULL;
    }

    if (bAllocNow) {
        m_pHashTable = new CAssoc* [nHashSize];
        memset(m_pHashTable, 0, sizeof(CAssoc*) * nHashSize);
    }
    m_nHashTableSize = nHashSize;
}
```

First, `InitHashTable()` checks if there is an existing hash table pointed to by `m_pHashTable`. If there is an existing table, `InitHashTable()` deletes it to make room for the new table. Next, if the `bAllocNow` flag is TRUE, `InitHashTable()` allocates an array of `CAssoc` pointers and sets `m_pHashTable` (a pointer to an array of `CAssoc` pointers) and zeroes out the memory using `memset`. Finally, `InitHashTable()` updates the `m_nHashTableSize` member so that values will not exceed the size of the internal hash array.

Finally, `InitHashTable()` updates the `m_nHashTableSize` member so that values will not exceed the size of the internal hash array.

Note: Knowing about `InitHashTable()` can be very handy. If you know that your map will have a certain size ceiling, for example 1017, you can go ahead and initialize the hash table yourself by calling the routine like this:

```
myMap.InitHashTable(1017, TRUE);
```

By doing this, you avoid subsequent memory allocation and manipulation of your map! But beware: calling this routine on a full map will entirely delete everything in the maps. Be sure that you are calling it at the right spot.

The default size of the hash table is 17. It always helps your hash distribution to have a prime-number size hash table, because it increases the distribution of the modulus operator.

The hash table is pretty straightforward: it's an array of CAssoc pointers. The next question you may have is How do associations get hashed into the table? The undocumented HashKey() member function is used to hash the key. The MFC maps have two different HashKey() functions, one for words and pointers and another for strings. Both HashKey() functions are shown in Listing 4-29.

Listing 4-29. The CMap hash function

```
inline UINT CMapWordToPtr::HashKey(WORD key) const
{
    return ((UINT)(void*)(DWORD)key) >> 4;
}
inline UINT CMapStringToString::HashKey(LPCTSTR key) const
{
    UINT nHash = 0;
    while (*key)
        nHash = (nHash<<5) + nHash + *key++;
    return nHash;
}
```

The first HashKey() routine is used to hash WORDs and void pointers. It shifts the accumulated hash value to increase the importance of the most significant upper bits.

The second HashKey() routine is used for strings. It iterates through the string and adds up a cumulative hash value, nHash, for each character in the string. It shifts to the left five bits to make sure the least significant, or most unique, bits of the string have greater importance and increase the distribution.

Both of these hash functions are designed to give a nice, even distribution for either WORDs/pointers or strings. Unfortunately, the nontemplate MFC map class designs don't let you provide your own HashKey() function. However, you can create your own hashing function with the template map classes.

Collisions!

One issue that usually pops up around hash tables is dealing with collisions. What if two values map to the same index in the hash table?

MFC maps take care of collisions by adding collided items to a linked list. In fact, if you look back at Listing 4-27, you will see that the CAssoc structure contains a

pNext pointer. CMaps use this pointer to store the associations that collide in an overflow-bucket linked list.

To see MFC map hashing in action, let's follow how an association is added to the map. All insertions go through the operator `[]`. The `SetAt()` member function directly calls this operator to insert a new value. Here's the inline `SetAt()` implementation, from `AFXCOLL.INL`:

```
void CMapWordToPtr::SetAt(WORD key, void* newValue)
{
    (*this)[key] = newValue;
}
```

`SetAt()` calls the `[]` operator for the key and then assigns the value to the result. To understand what's going on, let's take a look at `CMapWordToPtr::operator[]`. Listing 4-30 contains the pseudocode for the operator.

Listing 4-30. `CMapWordToPtr::operator[]`: The “key” to adding an association to a map

```
void*& CMapWordToPtr::operator[](WORD key)
{
    UINT nHash;
    CAassoc* pAssoc;
    if ((pAssoc = GetAssocAt(key, nHash)) == NULL) {
        if (_pHashTable == NULL)
            InitHashTable(_nHashTableSize);
        // it doesn't exist, add a new Association
        pAssoc = NewAssoc();
        pAssoc->nHashValue = nHash;
        pAssoc->key = key;
        // 'pAssoc->value' is a constructed object, nothing more
        // put into hash table
        pAssoc->pNext = _pHashTable[nHash];
        _pHashTable[nHash] = pAssoc;
    }
    return pAssoc->value; // return new reference
}
```

Operator `[]` serves the dual role of being the accessor and the storage helper for the hash table. From Listing 4-30, it's not obvious exactly where the hashing is taking place. It takes place in `GetAssocAt()`. We'll look at `GetAssocAt()` next, but first let's see what the operator is doing. The call to `GetAssocAt()` both sets `nHash` to the hashed index and checks to see if the entry already exists. If the association already exists in the hash table, the “if” block does not execute and the operator returns the value.

If the association is not already in the hash table, the operator first checks the hash table to make sure that it has been created and initialized. Next, operator [] creates a new association (CMap::NewAssoc() is exactly like the CList::NewNode() we covered in the list section) and stores the hash index and the key. At this point, notice that the value has not been set.

Finally, operator [] puts the new association into the hash table array at index nHash and returns a reference to the value pointer. Note that the last two lines of the “if” block will add the association to the collision buckets if there is already one there.

Operator [] returns a reference to the value, and the SetAt() member function actually stores the value.

CMapWordToPtr::GetAssocAt(): A Hashing Bottleneck

To get the full picture of what’s going on in the [] operator, we need to look at the GetAssocAt() routine, which locates an entry in the hash table. Listing 4-31 shows the undocumented GetAssocAt() member function.

Listing 4-31. CMapWordToPtr::GetAssocAt(): An undocumented map member that performs hashing for both lookup and storage

```
CMapWordToPtr::CAssoc* CMapWordToPtr::GetAssocAt(WORD key, UINT& nHash)
const
{
    nHash = HashKey(key) % m_nHashTableSize;
    if (m_pHashTable == NULL)
        return NULL;
    // see if it exists
    CAssoc* pAssoc;
    for (pAssoc = m_pHashTable[nHash]; pAssoc != NULL; pAssoc =
        pAssoc->pNext) {
        if (pAssoc->key == key)
            return pAssoc;
    }
    return NULL;
}
```

Before we dive into the source code, remember that GetAssocAt()’s job is to return NULL and a new hash index if the key does not exist. If the key does exist, GetAssocAt() should return the association for the key and also the hash index through the nHash argument.

In Listing 4-31, the first line of GetAssocAt() is the heart of the MFC map hash table. GetAssocAt() first calls HashKey() to run the hashing function for the key and then mods this with the size of the hash table to get the new hash index. All

accesses to the hash table except removing a key use this member function to hash the key.

Once a new hash index has been calculated, GetAssocAt() verifies that there is a valid initialized hash table and then retrieves the association that lives at the hash index. GetAssocAt() then iterates through the bucket list at the hash index to see if it can locate the key. If the key is found, the association is returned; if not, NULL is returned.

MFC Map Wrap

To summarize, we learned a couple of interesting points about MFC maps:

- They use the exact same CPlex memory management technique as MFC lists.
- An internal hash table gives maps very fast lookup of associations.
- You can specify the size of the hash table to improve the performance of maps that you use in your applications.

For More Information

One question that you might have after this review of MFC's map collections is Why does an association store the hash key? It seems that any code accessing the association would already know the key, so duplicate information is being stored. For the answer to this question, check out the CMapWordToPtr::GetNextAssoc() member function. Another interesting map member function is RemoveKey().

This concludes our coverage of the MFC collection classes. If you don't want to stop here, you might look at the template versions of the collections, which live in the AFXTEMP1.H header file.

The CFile Family: MFC Access to Files

MFC provides a hierarchy of utility classes that give the MFC programmer easy access to files. The root of the MFC file hierarchy, class CFile, implements basic nonbuffered file access that is a thin wrapper around the Windows file APIs. Class CStdioFile enhances CFile by providing buffered I/O. Because CStdioFile is buffered, reading and writing nonbinary ASCII files is much easier: you can look ahead in the buffer to read complete lines.

Class CMemFile provides a base class for implementing shared memory through a CFile interface. Unfortunately, CMemFile does not provide all of this functionality, but it does give you a head start.

An undocumented CFile derivative, CSharedFile, which is revealed later in this section, takes CMemFile the extra step and does provide some memory sharing. You can still use CMemFile if you want to serialize something to memory and then save it.

Using CFile

When you create a CFile, you specify a file name and a file open mode. You can either specify these in the constructor or defer opening the file and call the CFile::Open() member function with the file name and mode. Modes are combinations of access modes and sharing modes that are ORed together to achieve the desired results. Table 4-2 shows the various CFile modes available.

Table 4-2. An overview of the CFile access and sharing flags

Mode	Description
modeCreate	Creates the file. If the file already exists, sets its size to zero.
modeNoTruncate	Creates file, doesn't truncate.
modeRead	Opens the file for reading only.
modeReadWrite	Opens the file for both reading and writing.
modeWrite	Opens the file for writing.
modeNoInherit	Prevents the file from being inherited by child processes.
shareDenyNone	Opens the file without denying other processes read or write access.
shareDenyRead	Denies other processes read access to the file.
shareDenyWrite	Denies other processes write access to the file.
shareExclusive	Denies other processes both read and write access to the file.
shareCompat	Opens the file in compatibility mode, which allows any process to open the file any number of times.
typeText	Sets text mode with special processing for carriage-return pairs (used in CStdioFile only).
typeBinary	Sets binary mode (also only used in CStdioFile).

These modes are declared inside of class CFile, so you access them by specifying the CFile scope. For example: CFile::shareDenyRead, CFile::typeText, and so on.

The following line shows how to open a CFile in create and write mode.

```
CFile myFile ("myfile.sct", CFile::modeCreate | CFile::modeWrite);
```

CFile has a variety of member functions that let you read, write, lock, seek, rename, remove, and retrieve the status of the file.

CFile Internals

Because using CFile is so similar to using the standard Windows file APIs, let's go ahead and take a quick look at how it is implemented and then examine the more specialized derivatives.

The declaration for CFile lives in AFX.H along with most of the other MFC utility classes. The implementation for CFile is spread between two files: FILECORE.CPP contains all CFile member functions except for those that obtain file status; those live in FILEST.CPP.

The CFile Declaration

The abbreviated declaration of CFile, which highlights its undocumented implementation members, is in Listing 4-32.

Listing 4-32. An abbreviated CFile declaration

```
class CFile : public CObject
{
DECLARE_DYNAMIC(CFile)
public:
// Flag values **omitted
// Constructors **omitted
// Attributes **mostly omitted
    UINT m_hFile;
// Operations **omitted
// Overridables **omitted
// Implementation **omitted

public:
    virtual ~CFile();
    enum BufferCommand { bufferRead, bufferWrite, bufferCommit,
        bufferCheck };
    virtual UINT GetBufferPtr(UINT nCommand, UINT nCount = 0,
        void** ppBufStart = NULL, void** ppBufMax = NULL);
protected:
    BOOL m_bCloseonDelete;
    CString m_strFileName;
};
```

The first member of interest from the CFile declaration is the public `m_hFile` member, which stores the Windows file handle. CFile exposes this member so that CFile users can directly obtain the file handle if they want to use it in direct Windows

API calls. For example, CFile does not provide any Windows NT file security APIs, but you can call them by using CFile::m_hFile as the file handle.

The next interesting members are the BufferCommand enumeration and the GetBufferPtr() member function. In CFile, these members do nothing, but the CMemFile derivative does provide an implementation for GetBufferPtr(). This member is used by the MFC serialization mechanism to optimize serialization with a CMemFile.

Member m_bCloseOnDelete specifies if the file handle should be closed when the CFile is destroyed. Member m_strFileName is used to store the file name for the current file.

CFile Operations

Most of the CFile member functions call directly through to a Windows API, so there's not much value in looking at the source code for the members. Instead, Table 4-3 shows the various CFile member functions and the Windows APIs that they are based upon.

Table 4-3. A mapping of CFile member functions to Windows file APIs

CFile Member	Windows File API Wrapped
Open()	::Createfile()
Duplicate()	::DuplicateHandle()
Read()	::ReadFile()
Write()	::WriteFile()
Seek()	::SetFilePointer()
GetPosition()	::SetFilePointer()
Flush()	::FlushFileBuffers()
Close()	::CloseHandle()
LockRange()	::LockFile()
UnlockRange()	::UnlockFile()
SetLength()	::SetEndOfFile()
Rename()	::MoveFile()
Remove	::DeleteFile()
GetStatus()	::GetFileTime(), ::GetFileSize()

Interesting Undocumented Helper Functions

While looking in FILECORE.CPP, you might notice that there are some pretty handy global helper functions that help with file name manipulation. It's good to know about these functions, since they might save you time in your own file name manipulations.

- AfxResolveShortcut()—Looks up a Windows 95 shortcut and converts it into a full file name.
- AfxFullPath()—Turns a file path into an absolute path.
- AfxGetRoot()—Parses a Uniform Naming Convention (UNC) or old-style path and chops out the volume name.
- AfxComparePath()—Compares two paths to determine if they are the same. Handles all Double Byte CharacterSet (DBCS)/UNICODE issues for you.
- AfxGetFileName()—Parses out the name of a file from a path.

If you are interested in using one of these member functions, you should probably copy the code from FILECORE.CPP into your own functions, because these undocumented functions might change or be deleted in future MFC versions.

Because CFile is a wafer-thin abstraction of a file handle, it does not have many meaty internals. Class CStdioFile is more interesting because it adds buffering to CFile. Let's look at that CFile derivative and see how it supports file I/O buffering.

CStdioFile

Using a CStdioFile is very similar to using a CFile, except that it adds two new member functions: ReadString() and WriteString(). These two member functions read and write strings when the CStdioFile is open in text mode.

ReadString() reads until one of these conditions is true:

1. ReadString() grabs the specified number of characters.
2. ReadString() hits a carriage return/line feed pair.
3. ReadString hits the end of the file.

The following code fragment shows how to count the number of lines in a text file using CStdioFile.

```
int nLineCount = 0;
char buffer[256];
CStdioFile myFile("countme.txt", CFile::typeText | CFile::modeRead);
while (csfTipFile.ReadString(buffer, 256) != 0)
    nLineCount++;
```

Also, note that CStdioFile does not support some of the CFile functions because of its buffering. The unsupported CFile members include Duplicate(), LockRange(), and UnlockRange().

CStdioFile Internals

Externally, CStdioFile seems as if it adds only a couple of new members to CFile, but internally CStdioFile performs a good bit of extra logic to support the file buffering. While CFile wraps the Windows file APIs, CStdioFile wraps the C run-time library stream file functions such as fopen, fread, fputs, fgets, and fseek to get the file buffering and text operations. To use these functions, CStdioFile has to maintain a FILE pointer or stream pointer in addition to a CFile-inherited file handle. Let's look at the CStdioFile declaration and see where CStdioFile is storing this additional information.

The CStdioFile Declaration

Like CFile, CStdioFile's declaration lives in AFX.H. The implementation of CStdioFile can be found in FILETXT.CPP. Listing 4-33 contains an abbreviated CStdioFile declaration.

Listing 4-33. An abbreviated CStdioFile declaration

```
class CStdioFile : public CFile
{
DECLARE_DYNAMIC(CStdioFile)
public:
// Constructors **omitted
// Attributes
    FILE* m_pStream;      // stdio FILE
// m_hFile from base class is _fileno(m_pStream)
// Operations
    virtual void WriteString(LPCTSTR lpsz);
    virtual LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
    BOOL ReadString(CString& rString);
// Implementation
public:
    virtual ~CStdioFile();
    virtual DWORD GetPosition() const;
    virtual BOOL Open(LPCTSTR lpszFileName, UINT nOpenFlags,
                      CFileException* pError = NULL);
    virtual UINT Read(void* lpBuf, UINT nCount);
    virtual void Write(const void* lpBuf, UINT nCount);
    virtual LONG Seek(LONG lOff, UINT nFrom);
    virtual void Abort();
    virtual void Flush();
    virtual void Close();
// Unsupported APIs
    virtual CFile* Duplicate() const;
    virtual void LockRange(DWORD dwPos, DWORD dwCount);
    virtual void UnlockRange(DWORD dwPos, DWORD dwCount);
};
```

From the CStdioFile declaration, you can see that it keeps the FILE pointer in the `m_pStream` data member. The new member functions, `WriteString()` and `ReadString()`, have been included for your reference. Also notice that CStdioFile has quite a few implementation member functions that provide wrappers around the C run-time stream file calls.

ReadString()

To understand how CStdioFile works, let's look at the pseudocode for `CStdioFile::ReadString()` from `STRXTXT.CPP`, shown in Listing 4-34.

Listing 4-34. The `CStdioFile::ReadString()` implementation, from `STRXTXT.CPP`

```
LPTSTR CStdioFile::ReadString(LPTSTR lpsz, UINT nMax)
{
    LPTSTR lpszResult = _fgetts(lpsz, nMax, m_pStream);
    if (lpszResult == NULL && !feof(m_pStream))      {
        clearerr(m_pStream);
        AfxThrowFileException(CFileException::generic, _doserrno,
            m_strFileName);
    }
    return lpszResult;
}
```

From the implementation of `CStdioFile::ReadString()`, you can see that most CStdioFile operations wrap a C run-time call like `_fgetts` and then perform some extra error checking.

Table 4-4 shows the mapping of CStdioFile member functions and C run-time functions.

Table 4-4. A mapping of CStdioFile member functions and C run-time file stream calls

CStdio Member	Wrapped C Run-Time API
Open()	<code>_fdopen()</code>
Read()	<code>fread()</code>
Write()	<code>fwrite()</code>
ReadString()	<code>_fgetts()</code>
WriteString()	<code>_fputts()</code>
Seek()	<code>fseek()</code>
GetPosition()	<code>ftell()</code>
Flush()	<code>fflush()</code>
Close()	<code>fclose()</code>

So far, classes CFile and CStdioFile have not provided any juicy MFC internals. Let's move on to class CMemFile and its derivative, the undocumented CSharedFile class.

CMemFile

Because class CMemFile gives you a CFile interface to a chunk of memory, there is no need to provide a file name or maintain an `m_hFile` file handle.

CMemFile—the “Real” Story

Many MFC users think that class CMemFile gives you access to “shared” memory through Win32 memory-mapped files or some other shared memory mechanism. This is not correct. The memory used by CMemFile is standard memory and not shared. It is up to you to derive from CMemFile and add shared memory.

The main reason that MFC provides a CMemFile is so that you can serialize to a block of memory. (See Chapter 5 for more information on MFC serialization.) For example, you can use CArcive/CMemFile to generate a memory image of your objects, then write the memory image as binary data to the registry. Visual C++ uses this technique heavily.

The ability to use a CMemFile in serialization is an excellent example of object-oriented polymorphism. Since the serialization logic is using a CFile interface, CMemFile is none the wiser that it is writing to a file or to a block of memory as long as the interface behaves the same.

CMemFile automatically allocates, grows, and frees the memory block for you. You can allocate your own block of memory and Attach() a CMemFile to it if you wish.

CMemFile uses memory functions such as malloc(), realloc(), memcpy(), and free() to manage the internal block of memory. You can specify the size by which you want the memory block to grow through the `nGrowBytes` parameter in CMemFile's constructor. Overriding member functions like Alloc(), Free(), Realloc(), Memcpy(), and GrowFile() allows you to provide your own memory handling if you wish. The default implementations of these files call the equivalent C run-time memory routine.

Many CFile member functions do not make sense when applied to a block of memory, so they are not supported by CMemFile. These include Duplicate(), LockRange(), and UnlockRange().

Let's see how CMemFile provides the CFile interface to a block of memory.

CMemFile Internals

Unlike CFile and CStdioFile, CMemFile has some pretty cool internal details. Mapping a file interface to a block of memory is often a fairly tricky job.

The declaration for CMemFile is in AFX.H, and the implementation is contained in the FILEMEM.CPP MFC source file. The CMemFile::GetStatus() member function lives in FILEST.CPP.

The CMemFile Declaration

Listing 4-35 contains the abbreviated CMemFile declaration.

Listing 4-35. The CMemFile abbreviated declaration, from AFX.H

```
class CMemFile : public CFile
{
DECLARE_DYNAMIC(CMemFile) public:
// Constructors
    CMemFile(UINT nGrowBytes = 1024);
    CMemFile(BYTE* lpBuffer, UINT nBufferSize, UINT nGrowBytes = 0);
// Operations
    void Attach(BYTE* lpBuffer, UINT nBufferSize, UINT nGrowBytes = 0);
    BYTE* Detach();
// Advanced Overridables **omitted
protected:

// Implementation
protected:
    UINT m_nGrowBytes;
    DWORD m_nPosition;
    DWORD m_nBufferSize;
    DWORD m_nFileSize;
    BYTE* m_lpBuffer;
    BOOL m_bAutoDelete;

public:
    virtual ~CMemFile();
    virtual DWORD GetPosition() const;
    BOOL GetStatus(CFileStatus& rStatus) const;
    virtual LONG Seek(LONG loff, UINT nFrom);
    virtual void SetLength(DWORD dwNewLen);
    virtual UINT Read(void* lpBuf, UINT nCount);
    virtual void Write(const void* lpBuf, UINT nCount);
    virtual void Abort();
    virtual void Flush();
    virtual void Close();
    virtual UINT GetBufferPtr(UINT nCommand, UINT nCount = 0,
                           void** ppBufStart = NULL, void** ppBufMax = NULL);
// Unsupported APIs **omitted
};
```

CMemFile introduces quite a few new data members:

- **m_nGrowBytes**—Specifies the size CMemFile uses to grow the internal buffer whenever a new chunk of memory has to be allocated.
- **m_nPosition**—Specifies the current position of the simulated file pointer in the block of memory. This is manipulated by member functions such as Seek(). m_nPosition specifies where in the memory block reads and writes take place.
- **m_nBufferSize**—Specifies the size of the actual allocated buffer.
- **m_nFileSize**—Specifies the size of the logical memory file. This can be smaller than the buffer size due to the fact that memory is allocated in m_nGrowBytes chunks.
- **m_lpBuffer**—Represents the pointer to the actual memory of the memory file.
- **m_bAutoDelete**—Specifies if the memory should be freed when the destructor is called.

CMemFile Memory Handling

The CMemFile memory handling is actually very similar to the CArray memory handling in that a block of memory is managed and grown as needed. All of the CMemFile member functions first check if the operation is inside the bounds of the current block of memory. If not, they call the CMemFile::GrowFile() member function to either allocate or reallocate enough memory for the operation.

Listing 4-36 contains the pseudocode for the CMemFile::GrowFile() member function, from FILEMEM.CPP.

Listing 4-36. CMemFile::GrowFile() pseudocode, from FILEMEM.CPP

```
void CMemFile::GrowFile(DWORD dwNewLen)
{
    if (dwNewLen > m_nBufferSize)  {
        // grow the buffer
        DWORD dwNewBufferSize = (DWORD)m_nBufferSize;
        // determine new buffer size
        while (dwNewBufferSize < dwNewLen)
            dwNewBufferSize += m_nGrowBytes;
        // allocate new buffer
        BYTE* lpNew;
        if (m_lpBuffer == NULL)
            lpNew = Alloc(dwNewBufferSize);
        else
            lpNew = Realloc(m_lpBuffer, dwNewBufferSize);
        m_lpBuffer = lpNew;
        m_nBufferSize = dwNewBufferSize;
    }
}
```

`GrowFile()` takes an argument that specifies the desired new length of the buffer. If memory needs to be allocated, `GrowFile()` calculates the new memory size in `m_nGrowBytes`-divisible units. If the buffer has not been allocated already, `GrowFile()` calls `Alloc()`. If a buffer already exists, it calls `ReAlloc()` with the old pointer and the new desired size. Finally, all of the data members are updated to contain the new sizes of the memory block.

Accessing the Memory Block

To get a feel for what's involved in accessing the block of memory, let's look at the `CMemFile::Read()` member function implementation. Listing 4-37 contains the pseudocode version of the member function.

Listing 4-37. `CMemFile::Read(): Accessing the memory file`

```
UINT CMemFile::Read(void* lpBuf, UINT nCount)
{
    if (nCount == 0)
        return 0;
    if (m_nPosition > m_nFileSize)
        return 0;
    UINT nRead;
    if (m_nPosition + nCount > m_nFileSize)
        nRead = (UINT)(m_nFileSize - m_nPosition);
    else
        nRead = nCount;
    memcpy((BYTE*)lpBuf, (BYTE*)m_lpBuffer + m_nPosition, nRead);
    m_nPosition += nRead;
    return nRead;
}
```

First, `Read()` checks that the number of bytes to be read is nonzero and that the position of the internal pointer is not erroneously set (for example, larger than the size of the memory file).

Next, `Read()` calculates the number of bytes to be read by making sure that the number of bytes to read plus the current position is not larger than the block of memory (which would be like trying to read past the end of a file). If the count plus position is larger, `Read()` lowers the number of bytes to read to avoid reading past the block of memory.

Finally, `Read()` calls `Memcpy()` to copy the calculated number of bytes from the internal memory buffer into the destination memory buffer and returns the actual number of bytes read.

`CMemFile::Read()` is a good example of how the rest of the `CMemFile` operations are implemented. Externally they provide a file-like interface, while internally a block of memory is manipulated.

Advanced Topic: CFile::GetBufferPtr() and MFC Serialization

CMemFile provides a GetBufferPtr() routine that is used exclusively by the CArchive class covered in the next chapter. CArchive provides its own buffering scheme if necessary but first calls the CFile::GetBufferPtr() routine to see if the file supports buffering. If the CFile does support buffering, CArchive uses that buffering support instead of its own.

The CArchive communicates with the file by specifying one of the BufferCommand enumerations, which can be one of the following:

- bufferCheck—CArchive is asking the CFile if it directly supports buffering. CFile and CStdioFile answer no. CMemFile answers yes.
- bufferRead—CArchive is reading from the buffer. Specifies a start, a size, and a max.
- bufferWrite—CArchive is writing to the buffer.
- bufferCommit—CArchive wants to commit changes to the buffer.

With the different buffer commands in mind, let's look at the CMemFile::GetBufferPtr() implementation in Listing 4-38, from FILEMEM.CPP, to see how CMemFile and CArchive work together.

Listing 4-38. The Advanced CMemFile::GetBufferPtr() member function, from FILEMEM.CPP

```
UINT CMemFile::GetBufferPtr(UINT nCommand, UINT nCount, void** ppBufStart,
void** ppBufMax)
{
    if (nCommand == bufferCheck)
        return 1;
    if (nCommand == bufferCommit)      {
        // commit buffer
        m_nPosition += nCount;
        if (m_nPosition > m_nFileSize)
            m_nFileSize = m_nPosition; return 0;
    }

    // when storing, grow file as necessary to satisfy buffer request
    if (nCommand == bufferWrite && m_nPosition + nCount > m_nBufferSize)
        GrowFile(m_nPosition + nCount);
    // store buffer max and min
    *ppBufStart = m_lpBuffer + m_nPosition;

    // end of buffer depends on whether you are reading or writing
    if (nCommand == bufferWrite)
        *ppBufMax = m_lpBuffer + min(m_nBufferSize, m_nPosition + nCount);
}
```

```

    else
        *ppBufMax = m_lpBuffer + min(m_nFileSize, m_nPosition + nCount);
        m_nPosition += LPBYTE(*ppBufMax) - LPBYTE(*ppBufStart);
    }

    // return number of bytes in returned buffer space (may be <= nCount)
    return LPBYTE(*ppBufMax) - LPBYTE(*ppBufStart);
}

```

If GetBufferPtr() is called with the bufferCheck command, it returns 1, because direct buffering is supported. If the command is bufferCommit, then GetBufferPtr() updates the internal position to include the bytes that the CArchive has updated and now wants to commit.

If CArchive calls GetBufferPtr() using the command bufferWrite for writing, GetBufferPtr() grows the memory file if necessary and then returns pointers to the start and end of the memory block in the ppBufStart and ppBufMax arguments. Note that the m_nPosition on a write is not updated at this point. The CMemFile waits until a commit is issued by the CArchive in case the buffer user decides to “take back” the write.

If CArchive calls GetBufferPtr() using the command bufferRead, similar calculations are made, but the internal position is also updated to reflect the read operation.

GetBufferPtr() returns the difference in the start and end of the pointer addresses, which is the size of the buffer block.

After reading up on CArchive in Chapter 5, you may find it interesting to look at the code for CArchive::CArchive and see the CArchive side of this direct buffering.

There Is Another . . .

If you stuck to your MFC documentation, you would think that this was the end of the CFile discussion. But there is another CFile class that is completely undocumented: CSharedFile.

CSharedFile is a CMemFile derivative that uses the Windows global memory APIs, such as GlobalAlloc(), GlobalReAlloc(), GlobalLock(), and GlobalFree(), instead of the alloc/realloc/free memory handlers. This class is so clandestine that its declaration has been hidden away in the most sacred of MFC header files, AFXPRIV.H. The implementation for CSharedFile lives in the FILESHRD.CPP source file.

Listing 4-39 contains the declaration of CSharedFile from AFXPRIV.H.

Listing 4-39. The undocumented CSharedFile class declaration

```

class CSharedFile : public CMemFile
{
DECLARE_DYNAMIC(CSharedFile) public:

```

```

// Constructors
CSharedFile(UINT nAllocFlags = GMEM_DDESHARE|GMEM_MOVEABLE,
            UINT nGrowBytes = 4096);
// Attributes
HGLOBAL Detach();
void SetHandle(HGLOBAL hGlobalMemory, BOOL bAllowGrow = TRUE);
// Implementation
public:
    virtual ~CSharedFile(); protected:
    virtual BYTE* Alloc(DWORD nBytes);
    virtual BYTE* Realloc(BYTE* lpMem, DWORD nBytes);
    virtual void Free(BYTE* lpMem);
    UINT m_nAllocFlags;
    HGLOBAL m_hGlobalMemory;
    BOOL m_bAllowGrow;
};

}

```

First, notice that the default flags in the constructor specify GMEM_DDESHARE and GMEM_MOVEABLE, which denotes the global memory block as being able to be shared and moveable by the operating system.

Similar to the CMemFile::Attach() member function, CSharedFile provides a SetHandle() member function, which attaches a class to a previously allocated global memory handle.

Implementation-wise, CSharedFile is exactly like CMemFile, except that a global handle is used instead of a pointer.

Ok, What Is It Used For?

Now that we've revealed this private MFC class, the big question is "What is it used for?"

The only two instances where MFC uses CSharedFile are in the MFC OLE class source files OLEDOBJ1.CPP and OLEDOBJ2.CPP. In both of these instances it is used to read from a global handle that comes from the Clipboard and contains a chunk of memory to a picture (DIB).

If you ever find yourself with a global handle that's coming into your MFC application from legacy C code or somewhere like the Clipboard, you now know that you can attach a CSharedFile to the memory and access it through a CFile interface.

In fact, if you browse through some of the MFC samples (such as drawcli), you will see that Microsoft even uses this undocumented class in the samples!

CFile Summary

Classes CFile and CStdioFile provide thin MFC encapsulations over existing C file APIs. Class CMemFile provides a CFile interface to a block of memory. This is

especially useful if you want to share serialized data. Finally, we unearthed the CSharedFile, which is similar to CMemFile but provides a CFile interface to a block of memory created and referenced by a Windows global memory handle.

The last MFC utility class that we have on the agenda is the CException family of exception-handling classes.

CException: Providing Better Error Handling

To improve on the old C paradigm of checking a function's return value for an error, the C++ founding fathers came up with a more elegant solution called exceptions. Basically, you wrap any error-prone code in a “try” block and handle any errors that occur in a “catch” handler. When errors occur, they are thrown via the “throw” keyword, which causes the flow of execution to jump to the catch handler, which performs error handling and cleanup if necessary.

For example, Listing 4-40 checks for an exception when allocating memory.

Listing 4-40. Example of a C++ exception

```
//...
char * myPtr = NULL;
try
{
    myPtr = (char *)malloc(256);
    if (myPtr == NULL) //ERROR!!
        throw "Error, could not allocate memory!";
} //end try
catch(char * ptr) //Catch block for pointers
{
    TRACE(ptr);
    //Perform any cleanup here if necessary
}
```

Before version 2.0, MFC used to implement this language feature via macros, but now the MFC macros use the true C++ language exception handling. A remnant from this fact is that instead of using the C++ keywords directly, most MFC programmers still use the MFC exception-handling macros. These macros have a slightly different syntax than the C++ keywords. Table 4-5 shows the mapping of the C++ exception keyword to the MFC macros.

To clear up any confusion with these macros and keywords, let's look at how they are declared in AFX.H and in Listing 4-41.

Table 4-5. The mapping between the MFC exception macros and the C++ keywords

MFC Macro	C++ Keyword	Comments
TRY	try	—
CATCH	catch	—
AND_CATCH	catch	Used for second and additional catches
END_TRY	none	Marks the end of a try
THROW	throw	—
CATCH_ALL	catch(CException* e)	Catches all possible exceptions
THROW_LAST	throw	—

Listing 4-41. The MFC exception macros

```
#define TRY { AFX_EXCEPTION_LINK _afxExceptionLink; try {

#define CATCH(class, e) } catch (class* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
_afxExceptionLink.m_pException = e;
#define AND_CATCH(class, e) } catch (class* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
_afxExceptionLink.m_pException = e;

#define END_CATCH } }

#define THROW(e) throw e

#define THROW_LAST() (AfxThrowLastCleanup(), throw)
// Advanced macros for smaller code

#define CATCH_ALL(e) } catch (CException* e) \
{ { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e;

#define AND_CATCH_ALL(e) } catch (CException* e) \
{ { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e;

#define END_CATCH_ALL } } }

#define END_TRY } catch (CException* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e; } }
```

From Listing 4-41, you can tell that the MFC macros generally map to an `_afxExceptionLink` declaration, a C++ keyword, and some brackets for scoping.

MFC uses a structure called AFX_EXCEPTION_LINK to store and chain together catch blocks so that it can iterate through them and find the right handler that matches the type of the exception thrown.

Version Note

Later versions of MFC that use the C++ keywords do not need to keep track of the exceptions. However, it is up to you to delete the exception by calling `e->Delete()` on the exception.

By using these macros, you make sure that you use the MFC CException class, which has some enhancements over the vanilla C++ exception handlers.

Version Note

Before the language support for exceptions was supported by Visual C++, these macros used to actually call `setjmp/longjmp`. The old code is still maintained in current versions of the MFC to support older compilers. If you define `_AFX_OLD_EXCEPTIONS` and rebuild MFC, you will use the old-style version. These exceptions had the nasty side effect of not destroying any objects that were caught in a throw (called unwinding the stack). Depending on the implementation, this had the potential of allowing large memory leaks.

CException Internals

CException is an abstract base class that defines the interface for specialized CException derivatives that handle various types of exceptions. The exception classes and types of problems they handle in MFC are as follows:

- CArchiveException—Serialization exception.
- CDaoException—Data access object exception.
- CDBException—Database exception.
- CFileException—File exception.
- CMemoryException—Memory exception.
- CNotSupportedException—Something is not supported.
- COleDispatchException—OLE dispatch (Automation) exception.
- COleException—An OLE exception.
- CResourceException—Windows resource problem.
- CUserException—The end user caused an exception.

CException has two main member functions. GetErrorMessage() retrieves a buffer that contains a string describing the exception. ReportError displays a Windows message box with the exception string.

Internally, CException has only one undocumented data member, `m_bAutoDelete`. `m_bAutoDelete` determines if the CException will delete itself in the constructor.

Class CException is declared in AFX.H and its implementation is provided in the EXCEPT.CPP MFC source file. Because class CException is an abstract base class, there isn't very much interesting functionality to uncover. To get a better feel for the power of CExceptions, let's look at the CFileNotFoundException class, which handles CFile-related errors.

CFileNotFoundException

Looking at the declaration for CFileNotFoundException from AFX.H, you will see an enumeration that lists all of the possible reasons for a file exception:

- none—No error occurred.
- generic—An unspecified error occurred.
- fileNotFound—The file could not be located.
- badPath—All or part of the path is invalid.
- tooManyOpenFiles—The number of open files has been executed.
- accessDenied—The file could not be accessed.
- invalidFile—There was an attempt to use an invalid file handle.
- removeCurrentDir—The current working directory cannot be removed.
- directoryFull—No more directory entries allowed.
- badSeek—There was an error trying to set the file pointer.
- hardIO—There was a hardware error.
- sharingViolation—SHARE.EXE was not loaded, or a shared region was locked.
- lockViolation—There was an attempt to lock a region that was already locked.
- diskFull—The disk is full.
- endOfFile—The end of file was reached.

Before looking at some of the CFileNotFoundException internals from the source file FILEEX.CPP, let's see an example. Here's how a CFile derivative would use CFileNotFoundException to notify the user that one of the enumerated problems has occurred.

For example, the CMemFile::Seek() member function will throw a “badSeek” exception if the user tries to seek past the end of the internal memory block. When this case is detected, CMemFile::Seek() calls:

```
AfxThrowFileNotFoundException(CFileNotFoundException::badSeek);
```

AfxThrowFileException() is a global helper that is located in FILEX.CPP. The pseudocode for this routine is contained in Listing 4-42.

Listing 4-42. The global CFileException helper, AfxThrowFileException(), from FILEX.CPP

```
void AFXAPI AfxThrowFileException(int cause, LONG lOsError, LPCTSTR
    lpszFileName )
{
#ifdef _DEBUG
    LPCSTR lpsz;
    if (cause >= 0 && cause < _countof(rgszCFileExceptionCause))
        lpsz = rgszCFileExceptionCause[cause];
    else
        lpsz = szUnknown;
    TRACE3("Cfile exception: %hs, File %s, OS error information = %ld.\n",
        lpsz, (lpszFileName == NULL) ? "Unknown" : lpszFileName, lOsError);
#endif
    THROW(new CFileException(cause, lOsError, lpszFileName));
}
```

If the exception happens in a debug build, then AfxThrowFileException() prints out a helpful message with the exception type, file name, and even the operating system error number to the Visual C++ debug window using the TRACE3 macro.

After displaying the message, AfxThrowFileException() calls “throw,” triggering the exception, and passes in a new CFileException populated with the information about the error.

For example, we could have written the code segment that calls CMemFile::Seek() as shown in Listing 4-43.

Listing 4-43. Using CExceptions to catch errors

```
//...
CMemFile myMemFile;
try
{
    //...
    myMemFile.Seek(2043)
    //...
}
catch (CFileException, e)
{
    // If we get here, there was an error and e will contain
    // information about it.
    //Check for a seek problem
    if (e->m_cause == CFileException::badSeek)
        //uh-oh, a bad seek, do something...
    // Delete the exception now that it isn't needed
```

```
    e->Delete();  
}  
END_CATCH
```

There are three CFileException data members that help describe a CFile exception: m_cause contains one of the enumerated CFile problems; m_lOsError contains the Win32 error function returned by ::GetLastError(); and m_strFileName contains the name of the CFile in which the exception occurred. The CATCH section of any exception handler can access any of these methods to help handle the exception.

Other than that, CFileException contains some utility member functions for converting (1) between the enumerated problem types and strings and (2) from OS error codes to one of the enumerated CFileException exception causes.

Each of the CException derivatives is very similar to CFileException. They add information that describes the possible error situations and data members to hold any error state that might help the exception handler. They also usually have helpers for throwing the exception, which take as argument the exception state to be stored in the CException derivative.

The Undocumented Exception Class

There is one undocumented exception class. If you look in the AFX.H header file at the declaration of CMemoryException and CNotSupportedException, you will notice that contrary to what the MFC documentation shows, these two classes do not derive directly from CException, but from the undocumented CSimpleException class.

CSimpleException is a helper base class for both CMemoryException and CNotSupportedException that has room and some member functions for more storing of an error message.

For More Information

If you are interested in learning more about CSimpleException, you might want to look through EXCEPT.CPP to see what it adds to CException to try to figure out why CMemoryException and CNotSupportedException need to derive from it instead of CException.

Conclusion

In this chapter we looked at the MFC utility classes, such as the simple value types, the collections, file classes, and even exceptions. Along the way we uncovered some handy techniques, such as reference counting from CString and memory management

from CArray, that you can start using in your MFC classes immediately. We also learned about some pretty nifty undocumented classes, member functions, and data members. Some of these might be very useful if you find yourself in a bind. They are especially helpful if you ever try to derive your own class from one of the MFC utility classes.

In the next chapter, we tackle the root of the MFC hierarchy: the CObject. We look at the features it provides that form the cornerstone for the rest of MFC.

All Roads Lead to CObject

If you're like us and you've taped MFC hierarchies to the walls all around your work area, you've definitely noticed that one class is always popping up at the top of almost every class hierarchy: CObject. There's a very good reason for this. CObject defines and implements functionality that most MFC classes need in order to work with other parts of the framework. As you write MFC classes on your own or with Class Wizard, you should usually make sure that you have CObject somewhere in your class hierarchy to get these features.

Isn't That Expensive?

There's a lot of debate about whether it's good design to have almost every class in MFC derived from CObject—what C++ linguists call a singly rooted hierarchy. Some people argue that such a design causes the virtual tables to bloat and dramatically decreases run-time performance; still others argue that this is poor object-oriented design.

After we've unveiled all of CObject's internals, we'll revisit this question and compare the functionality that CObject delivers with the performance hit taken by adding CObject's virtual table entries.

CObject Features

Class CObject acts like the cookie cutter for all MFC classes. When you derive from CObject, your class automatically gains the ability to add four key MFC features:

- Run-time class information

- Dynamic creation
- Persistence (serialization)
- Run-time object diagnostics

The CObject features build on each other, so we'll take a look at how each feature works sequentially and then see how they all work together.

Before we dig deeper, here's the declaration for CObject. It's very important for every MFC internalist to be aware of what CObject functions are available and how they work. Listing 5-1 shows the declaration of CObject from AFX.H.

Listing 5-1. The root of the MFC tree: CObject's declaration

```
class CObject
{
public:
    virtual CRuntimeClass* GetRuntimeClass() const;
    virtual ~CObject(); // virtual destructors are necessary
// Diagnostic allocations
    void* operator new(size_t, void* p);
    void* operator new(size_t nSize);
    void operator delete(void* p);

#ifndef _DEBUG
    // for file name/line number tracking using DEBUG_NEW
    void* operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
#endif
protected:
    CObject();
private:
    CObject(const CObject& objectSrc); // no implementation
    void operator=(const CObject& objectSrc); // no implementation
// Attributes
public:
    BOOL IsSerializable() const;
    BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Overridables
    virtual void Serialize(CArchive& ar);
// Diagnostic Support
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
// Implementation
public:
    static AFX_DATA CRuntimeClass classCObject;
};
```

There are a couple of C++ tricks being used in CObject's declaration. First, the destructor has been made virtual so that the destructor can be called via polymorphism. In other words, MFC is going to be creating and deleting lots of CObject-derived classes, so they can delete a CObject without having to make sure the correct delete operator or destructor gets called.

Next, you'll notice that the copy constructor, CObject(const CObject& objectSrc), and the operator = are private. The MFC designers do this to force the compiler to generate an error if you try to assign one CObject to another via operator =, or attempt to create a CObject from another CObject using the copy constructor.

At this point, you can ignore macros like AFX_DATA and AFX_DATADEF. They are used for MFC's many DLL schemes and will be covered in detail in Chapter 10.

As we look at the features supplied by CObject, we'll be referring back to the CObject declaration. You may want to keep your thumb on Listing 5-1 so you can flip back and forth quickly.

An Introduction to Macros

One common pattern in MFC is the use of pairs of macros (DECLARE/IMPLEMENT) to add various features to your classes.

The DECLARE macros always declare some member variables and functions for a class, while the IMPLEMENT macros always implement the member functions. The DECLARE macros always go in the header file and the IMPLEMENT macros always go in the C++ file.

Whenever possible we expand the macros and see what makes them tick, but it is important to remember that the DECLARE macro is adding members to your class and the IMPLEMENT macros are providing the implementation, usually member functions, of what was declared in the DECLARE macro.

Run-Time Class Information

CObject's run-time class information (RTCI) feature lets the developer determine information about an object such as class name and parent at run time.

Even if you've been using MFC for some time, you might not realize that you are using RTCI in your classes. That's because the code to support RTCI lives in these macros:

```
DECLARE_DYNAMIC/IMPLEMENT_DYNAMIC
DECLARE_DYNCREATE/IMPLEMENT_DYNCREATE
DECLARE_SERIAL/IMPLEMENT_SERIAL
```

We'll cover the second and third sets of macros in other sections. Class Wizard automatically adds the declare macros to your class declaration (.H) file and the matching implement macro to your class implementation (.CPP) file.

Once you have added the mysterious macros to your CObject-derived class, you can call IsKindOf() to test the type of a class. This function takes an argument that is created by yet another macro, RUNTIME_CLASS.

Listing 5-2 shows an example of how to use MFC RTCI.

Listing 5-2. An example use of MFC RTCI

```
//This lives in .h file
class CMyClass : public CObject
{
    DECLARE_DYNAMIC(CMyClass); //Mystery macro 1
public:
    CMyClass();
    //...
}
//This lives in .cpp file
IMPLEMENT_DYNAMIC(CMyClass, CObject) //Mystery macro 2
DemoRTCI()
{
    CObject * pObject = new CMyClass;
    //Mystery macro 3
    if ( pObject->IsKindOf(RUNTIME_CLASS(CMyClass))){
        CMyClass * pMyObject = (CMyClass *)pObject;
    }
}
```

This example shows how to use RTCI to check that a polymorphic cast to a derived class is safe.

Why Not Use C++ RTTI?

At this point you may be asking why the MFC developers did not just use the C++ run-time type information language (RTTI) feature instead of going to all this trouble with macros. The MFC team is separate from the language team, and they needed RTTI support in the very early days of MFC. Instead of waiting on the language group to add C++ RTTI support (which they finally did in version 4 of Visual C++), the MFC team implemented their own support. Rumor has it that a future version of MFC will replace the implementation of the RTCI macros to use the C++ RTTI feature. This won't affect most applications because the macros isolate them from change.

RTCI: How Does It Work?

Now that you've seen an example of how to use RTCI, let's take a look at how the MFC team implemented this support and see if there are any interesting side effects that you can use in your programs.

The RTCI macros can be found in AFX.H, and the source code for RTCI support is located in the MFC source file OBJCORE.CPP. Let's start by dissecting the macros and follow the trail from there.

When you place the DECLARE_DYNAMIC macro in your class header, you pass it the name of the class as an argument. Let's look at the definition of DECLARE_DYNAMIC and then apply the macro to a sample class name.

Here's the declaration of the DECLARE_DYNAMIC macro:

```
#define DECLARE_DYNAMIC(class_name) \
public: \
static CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
```

One trick this macro employs is the preprocessor concatenation operator “##”. Operator ## tells the preprocessor to concatenate what's on the right of the operator onto what's on the left. For example, *MFC##Internals* would generate *MFCInternals*. If arguments appear in the concatenation, they are replaced and then concatenated. Get used to this. MFC uses this technique all over the place.

Plugging CMyClass into the DECLARE_DYNAMIC macro and running the preprocessor causes the following code to be added to your CObject-derived class definition:

```
public: \
static CRuntimeClass classCMyClass;
virtual CRuntimeClass * GetRuntimeClass() const;
```

The first line creates a static CRuntimeClass member called classCMyClass. Recall how static class members work: no matter how many instances of a class you create, there is only one copy of the static method.

If you look back at the CObject declaration in Listing 5-1, you'll see that the CMyClass::GetRuntimeClass() member function declaration overrides the virtual CObject::GetRuntimeClass(). This way, CMyClass::GetRuntimeClass() returns the correct run-time class information.

Another Class?

As you can tell, there is another class at work here: CRuntimeClass. Before looking at the IMPLEMENT portion of this macro, let's look at the static CRuntimeClass declaration in more detail.

Listing 5-3 highlights the declaration of CRuntimeClass from AFX.H:

Listing 5-3. The CRuntimeClass helper declaration

```
struct CRuntimeClass
{
// Attributes
    LPCSTR m_lpszClassName;
    int m_nObjectSize;
    UINT m_wSchema; // schema number of the loaded class
    //Use to be m_pfnConstruct in MFC 2.x
    void (PASCAL* m_pfnCreateObject)(void* p);
    // if NULL => abstract base class
    CRuntimeClass* m_pBaseClass;
// Operations
    CObject* CreateObject();
// Implementation
    BOOL ConstructObject(void* pThis);
    void Store(CArchive& ar);
    static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);
    // CRuntimeClass objects linked together in simple list
    CRuntimeClass* m_pNextClass;      // linked list of registered classes
// special debug diagnostics
#ifndef _DEBUG
    BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
#endif
};
```

Stop the presses! This isn't a class at all. It's just a lowly structure! The name "CRuntimeClass" appears to be an MFC misnomer. Don't be alarmed. C++ structures are just classes in which everything defaults to public. In other words, structures can have member functions and other class-like features, but the default member scope is public so that you don't have to write:

```
class CMyClass {
public:
    // stuff
};
```

Because CRuntimeClass is a structure, all of its members are public. This is handy because it enables you to easily get to all the interesting CRuntimeClass members from your programs.

IMPLEMENT_DYNAMIC Exposed

To understand more about CRuntimeClass, let's look at what the IMPLEMENT_DYNAMIC(classname, base_classname) macro really does:

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
    AFX_DATADEF CRuntimeClass class_name::class##class_name = { \
        #class_name, sizeof(class_name), wSchema, pfnNew, \
        RUNTIME_CLASS(base_class_name), NULL }; \
    static const AFX_CLASSINIT \
        _init_##class_name(&class_name::class##class_name); \
    CRuntimeClass* class_name::GetRuntimeClass() const \
    { return &class_name::class##class_name; } \
```

Even though we've cleaned it up somewhat, such a macro can be a bit intimidating. There's another interesting preprocessor macro in use here: the "#", or string-izing macro. This macro turns its right side into a quoted string. Let's run **IMPLEMENT_DYNAMIC** through the preprocessor with the arguments **CMyClass** and **CObject** and shed some light on what this mess is really doing.

```
AFX_DATADEF CRuntimeClass CMyClass::classCMyClass = {
    "CMyClass", sizeof(CMyClass), 0xFFFF, NULL, RUNTIME_CLASS(CObject),
    NULL };
static const AFX_CLASSINIT _init_CMyClass(&CMyClass::classCMyClass);
CRuntimeClass * CMyClass::GetRuntimeClass() const
{
    return &CMyClass::classCMyClass;
}
```

Once you see the generated code, you can tell that this macro is doing three things:

1. First, **IMPLEMENT_DYNAMIC** initializes the static **CRuntimeClass** data member **classCMyClass** as follows (you may want to review the **CRuntimeClass** declaration again):

```
m_lpszClassName = "CMyClass";
m_nObjectSize = sizeof(CMyClass);
m_wSchema = 0xFFFF;
m_pfnCreateObject = NULL;
m_pBaseClass = RUNTIME_CLASS(CObject);
```

2. Second, **IMPLEMENT_DYNAMIC** creates a static **AFX_CLASSINIT** structure.

The second snippet of code generated by **IMPLEMENT_DYNAMIC** is confusing until you look at the declaration of **AFX_CLASSINIT**, which is a structure with only a constructor:

```
struct AFX_CLASSINIT
{ AFX_CLASSINIT(CRuntimeClass* pNewClass); };
```

So **IMPLEMENT_DYNAMIC** creates a static AFX_CLASSINIT structure and calls its constructor with the CMyClass::classCMYCLASS CRuntimeClass static structure. This causes the class to be added to an internal MFC state list, which we will cover in Chapter 9. (We'll also see more about this statement later in the "Memory Diagnostics" section toward the end of this chapter.)

3. Finally, the **IMPLEMENT_DYNAMIC** macro generates the overridden member function GetRuntimeClass(). GetRuntimeClass() just returns the address of the static CRuntimeClass member classCMYCLASS.

The last macro to dissect for runtime class information is **RUNTIME_CLASS**, whose definition is as follows:

```
#define RUNTIME_CLASS(class_name) (&class_name::class##class_name)
```

The **RUNTIME_CLASS** macro returns the static CRuntimeClass member data, which was placed in our class by the **DECLARE_DYNAMIC** macro (just like **GetRuntimeClass()**).

Inside **CObject::IsKindOf()**

This discussion brings up some interesting questions about CRuntimeClass. What are members like **m_wSchema** and **m_pfnCreateObject** used for? Why does it appear that CRuntimeClass has a linked list? What could it be used for? To help answer some of these gnawing questions, let's look at how **CObject::IsKindOf()** is implemented.

Listing 5-4 shows pseudocode for **CObject::IsKindOf()**, which lives in **OBJCORE.CPP**.

Listing 5-4. The implementation of **CObject::IsKindOf()**

```
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
    CRuntimeClass* pClassThis = GetRuntimeClass();
    while (pClassThis != NULL){
        if (pClassThis == pClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;           // walked to the top, no match
}
```

IsKindOf() takes the static CRuntimeClass pointer returned by **RUNTIME_CLASS** and then calls **GetRuntimeClass()** for the current object (or this). It then compares the CRuntimeClass pointers to see if they are pointing to the same static structure.

If they are the same object, a match has been made. If not, IsKindOf() walks up the inheritance tree looking for a match.

Notice the call to IsKindOf() in Listing 5-2. In this case, IsKindOf() starts at CMyClass, fails to find a match, then walks up the inheritance tree one level from CMyClass to CObject to find the match.

Check Listing 5-1 again. The CObject run-time class classCObject is declared at the bottom of the class. You may have missed it initially because it is not hidden by a DECLARE_DYNAMIC macro.

CRuntimeClass's Undocumented Function: IsDerivedFrom()

There are lots of functions we haven't covered that live in CRuntimeClass. We'll cover those functions and answer some of the CRuntimeClass questions later. But for now let's take a quick look at a specific undocumented function, IsDerivedFrom().

Listing 5-5 shows the implementation for CRuntimeClass::IsDerivedFrom().

Listing 5-5. Does this remind you of something?

```
BOOL CRuntimeClass::IsDerivedFrom(const CRuntimeClass* pBaseClass) const
{
    const CRuntimeClass* pClassThis = this;
    while (pClassThis != NULL){
        if (pClassThis == pBaseClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;           // walked to the top, no match
}
```

Look familiar? Yep, you guessed it, this is exactly the same as IsKindOf(), but you can call it through a CRuntimeClass object. MFC uses CRuntimeClass::IsDerivedFrom() to double-check that the developer has passed in a valid derived object (for example, MFC checks for a valid CWnd derivative in splitter windows). In many of these cases, an object hasn't been created yet, but MFC still needs to know if it is of the correct type based on a CRuntimeClass pointer. This undocumented function comes in very handy if you find yourself in a similar situation.

Let's move on to the CObject dynamic creation support and in the process learn more about the mysterious CRuntimeClass structure.

Dynamic Creation

Dynamic creationism is something you'll learn about only here—it's not taught in public schools. Just a joke. To add dynamic creation support to your CObject derivative,

you use the `DECLARE_DYNCREATE/IMPLEMENT_DYNCREATE` macros. These macros build on top of the `DECLARE_DYNAMIC/IMPLEMENT_DYNAMIC` RTCI macros, so there is no need to have both in your classes. In fact, using them together will generate a compiler error.

Once you have added these macros, you can create objects based on their CRuntimeClass information. To do so, call the CRuntimeClass member function `CreateObject()`.

Listing 5-6 illustrates an example of dynamic creation. This code assumes that `CMyClass` is similar to Listing 5-2 but uses the `DYNCREATE` macros instead of the `DYNAMIC` macros.

Listing 5-6. An example of how to use dynamic creation

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS( CMyClass );
CObject* pObject = pRuntimeClass->CreateObject();
ASSERT( pObject->IsKindOf( RUNTIME_CLASS( CMyClass ) ) );
```

MFC uses dynamic creation extensively. One example you may be familiar with is the MFC `CMultiDocTemplate` class that is created like this:

```
pDocTemplate = new CMultiDocTemplate(IDR_SCRIBTYPE,
    RUNTIME_CLASS(CScribDoc),
    RUNTIME_CLASS(CMDIChildWnd),
    RUNTIME_CLASS(CScribView));
```

MFC stores the CRuntimeClass structures and later instantiates objects using `CRuntimeClass::CreateObject()`. (For more information on document templates and how they work, see Chapter 7.)

How Does Dynamic Creation Work?

`CObject` Dynamic creation support is easy to understand once you understand how RTCI is implemented. Let's investigate the dynamic creation macros to see how dynamic creation works.

The key to dynamic creation is the `CRuntimeClass` structure and its `CreateObject()` method. The `DECLARE_DYNCREATE` macro output differs from the `DECLARE_DYNAMIC` macro by only one line:

```
static CObject * CreateObject();
```

As you can guess, `IMPLEMENT_DYNCREATE` generates the `CreateObject()` member function for the class in addition to the usual `IMPLEMENT_DYNAMIC` output. The generated `CreateObject()` implementation is fairly simple:

```
CObject* PASCAL CMyClass::CreateObject()
{
    return new CMyClass;
}
```

`IMPLEMENT_DYNCREATE` also invokes the `_IMPLEMENT_RUNTIMECLASS` utility macro with a slightly different argument. It passes in `CMyClass::CreateObject()` to initialize the `CRuntimeClass::m_pfnCreateObject` function pointer. The static `CRuntimeClass` structure is initialized with a pointer to `CMyClass::CreateObject()` in the `m_pfnCreateObject` member and uses `m_pfnCreateObject` to dynamically create an object of type `CMyClass` at run time.

The `CRuntimeClass::CreateObject()` Implementation

The implementation for `CreateObject()` lives in `OBJCORE.CPP`; a pseudocode version is in Listing 5-7.

Listing 5-7. `CRuntimeClass::CreateObject`, the heart of MFC dynamic creation

```
CObject* CRuntimeClass::CreateObject()
{
    if (m_pfnCreateObject == NULL) {
        //Error! User did not use correct macro
        return NULL;
    }
    CObject* pObject = NULL;
    TRY {
        pObject = (*m_pfnCreateObject)();
    }
    END_TRY
    return pObject;
}
```

`CRuntimeClass::CreateObject()` first checks to make sure that the user has specified the correct macros. `CreateObject()` then calls the function pointer stored in the `CRuntimeClass::m_pfnCreateObject` member variable. In our example, this points to `CMyClass::CreateObject()`, which in turn calls “`new CMyClass`”. The new object is passed back through the `CreateObject()` calls to the `CRuntimeClass::CreateObject()` caller.

What's Left to Learn about `CRuntimeClass`?

At this point we know what each `CRuntimeClass` member data and function is used for except these:

```
LPCSTR m_lpszClassName;
int m_nObjectSize;
```

```

UINT m_wSchema;
void Store(CArchive& ar) const;
static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);
CRuntimeClass* m_pNextClass;           // linked list of registered classes

```

To shed some light on these CRuntimeClass members, let's look at how MFC supports persistence.

Persistence in MFC

Persistence is the ability to store objects and recover their states some time later. This feature is referred to as *serialization* in MFC. Using persistence, you can quickly implement file reading and writing without having to worry about the format of the file you are writing to. You simply keep concatenating objects onto the file and then read them in the order they were written.

Looking back at the CObject declaration in Listing 5-1, you'll notice that CObject defines two member functions that relate to serialization: IsSerializable() and Serialize(). Let's quickly review how to add serialization to your classes and then dissect how MFC implements serialization.

Adding Serialization to Your Classes

To support serialization in your CObject-derived classes, you do two things:

1. Use the DECLARE_SERIAL/IMPLEMENT_SERIAL macros. The IMPLEMENT_SERIAL macro takes a WORD that defines the schema, or version information, for your object. This lets you tag versions of your object format. If the schemas do not match when the data is read, MFC will not read in your object.
2. Override the CObject::Serialize() method to read/write your member data.

Here's the new CMyClass declaration, with serialization support added, plus some new members to illustrate serialization.

```

class CMyClass : public CObject
{
    DECLARE_SERIAL(CMyClass);
public:
    CMyClass();
    WORD    m_wType;

```

```

    DWORD m_dwData;
    CPoint m_ptMiddle;
    void Serialize(CArchive &);

}

```

To implement the CMYClass::Serialize() member function, use the C++ insertion (<<) and extraction (>>) operators as follows:

```

IMPLEMENT_SERIAL(CMyClass, CObject, 0xabcd)
void CMYClass::Serialize(CArchive & ar)
{
    if (ar.IsStoring()){
        ar << m_wType;
        ar << m_dwData;
        ar << m_ptMiddle;
    }
    else{
        ar >> m_wType;
        ar >> m_dwData;
        ar >> m_ptMiddle;
    }
}

```

Serialization couldn't be easier! For basic types, you just use the extraction/insertion operators. For more complex types you call their Serialize() method, being sure to pass the CArchive argument to them.

Of course, whenever something is easy, there is a lot of work going on behind the scenes. This is especially true of MFC serialization!

Serialization: How It Works

In this section we'll point out serialization declarations that live in both AFX.H and AFXWIN.H. The implementation of serialization can be found in the MFC source files ARCCORE.CPP and ARCOBJ.CPP.

To fully understand MFC serialization, we'll first look at the serialization helper class, CArchive. Then we will dissect the DECLARE_SERIAL/IMPLEMENT_SERIAL macros. Finally, we will trace both the storing and reading of a class from start to finish through the framework to see how serialization works chronologically.

The CArchive Helper Class

Class CArchive implements a binary stream instead of the more common ASCII C++ stream (or input/output stream). It is tied to a CFile pointer and implements buffering of data for better performance.

Let's look at how class CArchive is implemented. The definition for class CArchive lives in AFX.H and is shown in Listing 5-8.

Listing 5-8. The somewhat intimidating declaration of CArchive

```
class CArchive
{
public:
    // Flag values
    enum Mode { store = 0, load = 1, bNoFlushonDelete = 2, bNoByteSwap = 4 };
    CArchive(CFile* pFile, UINT nMode, int nBufSize = 512, void* lpBuf =
        NULL);
    ~CArchive();
// Attributes
    BOOL IsLoading() const;
    BOOL IsStoring() const;
    BOOL IsByteSwapping() const;
    BOOL IsBufferEmpty() const;
    CFile* GetFile() const;
    UINT GetObjectSchema(); // only valid when reading a CObject*
    void SetObjectSchema(UINT nSchema);
    CDocument* m_pDocument;
// Operations
    UINT Read(void* lpBuf, UINT nMax);
    void Write(const void* lpBuf, UINT nMax);
    void Flush();
    void Close();
    void Abort(); // close and shutdown without exceptions
// reading and writing strings
    void WriteString(LPCTSTR lpsz);
    LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
    BOOL ReadString(CString& rString);

public:
    friend CArchive& operator<<(CArchive& ar, const CObject* pOb);
    friend CArchive& operator>>(CArchive& ar, CObject*& pOb);
    friend CArchive& operator>>(CArchive& ar, const CObject*& pOb);
// insertion operations
    CArchive& operator<<(BYTE by);
    CArchive& operator<<(WORD w);
    CArchive& operator<<(LONG l);
    CArchive& operator<<(DWORD dw);
    CArchive& operator<<(float f);
    CArchive& operator<<(double d);
    CArchive& operator<<(int i);
    CArchive& operator<<(short w);
    CArchive& operator<<(char ch);
```

```

// extraction operations
CArchive& operator>>(BYTE& by);
CArchive& operator>>(WORD& w);
CArchive& operator>>(DWORD& dw);
CArchive& operator>>(LONG& l);
CArchive& operator>>(float& f);
CArchive& operator>>(double& d);
CArchive& operator>>(int& i);
CArchive& operator>>(short& w);
CArchive& operator>>(char& ch);
// object read/write
COBJECT* ReadObject(const CRuntimeClass* pClass);
void WriteObject(const COBJECT* pOb);
// advanced object mapping (used for forced references)
void MapObject(const COBJECT* pOb);
// advanced versioning support
void WriteClass(const CRuntimeClass* pClassRef);
CRuntimeClass* ReadClass(const CRuntimeClass* pClassRefRequested = NULL,
                        UINT* pSchema = NULL, DWORD* pObTag = NULL);
void SerializeClass(const CRuntimeClass* pClassRef);
// advanced operations (used when storing/loading many objects)
void SetStoreParams(UINT nHashSize = 2053, UINT nBlockSize = 128);
void SetLoadParams(UINT nGrowBy = 1024);
// Implementation
public:
    BOOL m_bForceFlat; // for ColeClientItem implementation (default TRUE)
    BOOL m_bDirectBuffer; // TRUE if m_pFile supports direct buffering
    void FillBuffer(UINT nBytesNeeded);
    void CheckCount(); // throw exception if m_nMapCount is too large
// special functions for reading and writing (16-bit compatible) counts
    DWORD ReadCount();
    void WriteCount(DWORD dwCount);
// public for advanced use
    UINT m_nObjectSchema;
    CString m_strFileName;
protected: // archive objects cannot be copied or assigned
    CArchive(const CArchive& arSrc);
    void operator=(const CArchive& arSrc);
    BOOL m_nMode;
    BOOL m_bUserBuf;
    int m_nBufSize;
    CFile* m_pFile;
    BYTE* m_lpBufCur;
    BYTE* m_lpBufMax;
    BYTE* m_lpBufStart;
// array/map for COBJECT* and CRuntimeClass* load/store
    UINT m_nMapCount;
    union {
        CPtrArray* m_pLoadArray;

```

```

    CMapPtrToPtr* m_pStoreMap;
};

// map to keep track of mismatched schemas
CMapPtrToPtr* m_pSchemaMap;
// advanced parameters (controls performance with large archives)
UINT m_nGrowSize;
UINT m_nHashSize;
};

```

Whoa! And you thought we were exaggerating when we said that behind everything easy there's something complex going on behind the scenes.

CArchive Is Largely Undocumented

If you look in your MFC reference manual, you will notice that CArchive has little documentation. Here's an overview of what some of the less-obvious members in CArchive are used for:

- **m_nMode**—Specifies if the archive is reading or writing. This can also be used to turn off flushing on delete. The Macintosh version of MFC uses the bNoByteSwap mode to automatically byte-swap during serialization.
- **IsLoading()**, **IsStoring()**, **IsByteSwapping()**, and **IsBufferEmpty()**—These accessors just access the mode to determine the state of the CArchive object.
- **Operators**—CArchive defines insertion and extraction operators for CObject-derived classes, the Windows data types, and the C++ types.
- **Map members**—When writing, CArchive maintains a map of items stored so that it can quickly access the class information of similar objects. CArchive also uses the map to ensure that it will write out CRuntimeClass information only once for a certain class then reference it later in the serialization stream.

When reading, CArchive maintains an array of objects already created and stores the already-read CRuntimeClass structure information in the array. This way, CArchive can look it up when it finds a reference that was written in the serialization stream.

The members **MapObject()**, **m_nMapCount**, **m_pLoadArray**, **m_pStoreMap**, **m_pSchemaMap**, and **m_nGrowSize** are all involved in implementing the map and arrays for reading and writing.

- **Class member functions**—The **WriteClass()**, **ReadClass()**, and **SerializeClass()** member functions serialize CRuntimeClass structure information. (More on this in a minute.)

The other methods implement fairly standard buffering and store information about the CFile that the CArchive is using.

Inside a CArchive Operator

To understand a little bit more about what CArchive is doing when we use it, let's look at the implementation of the extraction and insertion operators for WORD. These operators are inline and live in the INCLUDE\AFX.INL inline file.

Here's the implementation of the insertion operator:

```
_AFX_INLINE CArchive& CArchive::operator<<(WORD w)
{
    if (m_lpBufCur + sizeof(WORD) > m_lpBufMax)
        Flush();
    *(WORD*)m_lpBufCur = w;
    m_lpBufCur += sizeof(WORD);
    return *this;
}
```

All the insertion operator does is check if it needs to flush the internal CArchive buffer, place the WORD in the buffer, and finally increment the buffer pointer. The operator returns a CArchive reference so that insertion operators can be cascaded.

Let's look at the CArchive extraction operator for a WORD:

```
_AFX_INLINE CArchive& CArchive::operator>>(WORD& w)
{
    if (m_lpBufCur + sizeof(WORD) > m_lpBufMax)
        FillBuffer(sizeof(WORD) - (UINT)(m_lpBufMax - m_lpBufCur));
    w = *(WORD*)m_lpBufCur;
    m_lpBufCur += sizeof(WORD);
    return *this;
}
```

The WORD extraction operator checks to see if it needs to grab more data from the file into the internal buffer. Then it places a WORD's worth of data from the buffer into the word reference argument. Finally, it increments the buffer pointer to prepare for the next extraction.

Serialization Macros

Now that we have a pretty good feel for what CArchive does, let's look at what the serialization macros **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** are doing.

Here's the definition of **DECLARE_SERIAL**:

```
#define DECLARE_SERIAL(class_name) \
DECLARE_DYNCREATE(class_name) \
friend CArchive& operator>>(CArchive& ar, class_name* &pOb);
```

The DECLARE_SERIAL macro is really only adding one line of code to what we have already covered in the RTCI and dynamic creation sections. Here's what it will look like in your class declaration after the macro has been preprocessed:

```
friend CArchive& operator>> (CArchive& ar, CMyClass* &pOb);
```

This line declares a global extraction operator as a friend to your class.

Looking at IMPLEMENT_SERIAL reveals more about why:

```
#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
    CObject* class_name::CreateObject() \
    { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
    class_name::CreateObject) \
    CArchive& operator>>(CArchive& ar, class_name* &pOb) \
    { pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
    return ar; } \
```

Note that the IMPLEMENT_SERIAL macro is passing a new argument into the _IMPLEMENT_RUNTIMECLASS utility macro. The new argument is wSchema, which gets placed into the CRuntimeClass::m_wSchema member. (More on this later.)

IMPLEMENT_SERIAL generates the implementation of the global extraction macro:

```
CArchive& operator>>(CArchive& ar, CMyClass * &pOb)
{
    pOb = (CMyClass*) ar.ReadObject(RUNTIME_CLASS(CMyClass));
    return ar;
}
```

IMPLEMENT_SERIAL is overloading operator >> for the class so that it calls CArchive::ReadObject() with the static CRuntimeClass member returned by RUNTIME_CLASS. Let's follow the trail through ReadObject().

CArchive::ReadObject()

This is a pretty complex function. Listing 5-9 contains a pseudocode version that highlights the logic we are after.

Listing 5-9. The CArchive::ReadObject() member function implementation

```
CObject* CArchive::ReadObject(const CRuntimeClass* pClassRefRequested)
{
    UINT nSchema;
```

```

DWORD obTag;
CRuntimeClass* pClassRef = ReadClass(pClassRefRequested, &nSchema,
                                      &obTag);
// allocate a new object based on the class just acquired
pOb = pClassRef->CreateObject();
// Serialize the object with the schema number set in the archive
UINT nSchemaSave = m_nObjectSchema;
m_nObjectSchema = nSchema;
pOb->Serialize(*this);
m_nObjectSchema = nSchemaSave;
return pOb;
}

```

In CArchive::ReadObject(), first the function reads in the CRuntimeClass structure from the file. If a class of this type has already been read, there will be a reference, which ReadObject() looks up in the internal array. Next, ReadObject() creates an object using dynamic creation. Finally, ReadObject() calls the object's Serialize() method and passes in the current CArchive reference (pOb->Serialize(*this)).

Why No Insertion Operator?

At this point, you may be asking why the serialization macros don't define insertion operators as well as extraction operators. There is a global insertion operator defined in AFX.INL:

```

CArchive& operator<<(CArchive& ar, const CObject* pOb)
{
    ar.WriteObject(pOb);
    return ar;
}

```

WriteObject() is similar to ReadObject(). It calls WriteClass(), which writes out the CRuntimeClass information, and then WriteObject() calls the object's Serialize() member function.

Since WriteObject() does not need any specific CRuntimeClass information, the CObject insertion operator is sufficient and a new insertion operator for your CObject derivate does not need to be declared and implemented by the serialization macros.

CObject and Serialization

Now that we fully understand what the serialization macros are doing, let's go back to those two CObject serialization functions: IsSerializable() and Serialize().

CObject::IsSerializable()

The implementation of this function lives in OBJCORE.CPP:

```
BOOL CObject::IsSerializable() const
{
    return (GetRuntimeClass()->m_wSchema != 0xffff);
}
```

This function is just making sure that the user has specified the correct macros for serialization by checking that m_wSchema for the CRuntimeClass structure is not 0xffff.

The MFC manuals routinely caution that you should call the overiden CObject::Serialize() member function in your override before performing your class-specific serialization. Let's see why.

CObject::Serialize()

Surprisingly, the implementation for CObject::Serialize() literally does nothing! It is located in AFX.INL, if you want to check for yourself. This is just an empty placeholder for your Serialize() routines in the current versions of MFC . However, Microsoft may very well change this and add some code that must be called before your class-specific Serialize() code in future versions of MFC. While it's good form to call CObject::Serialize(), now you know that it really isn't doing anything (yet).

Tracing a Write and a Read of a CObject Derived Object

To help understand how serialization works, let's trace a complete write and read operation as it progresses through the framework. To make it interesting, let's assume that we are using the document/view architecture (covered in detail in Chapter 7) and that our document contains a CMYClass member pointer called m_pMyClass. Here's what the Serialize() method of our vanilla CDocument derivative, CMYDocument, looks like:

```
void CMYDocument::Serialize(CArchive & ar)
{
    if (ar.IsStoring()){
        ar << m_pMyClass ;
    } else {
        ar >> m_pMyClass;
    }
}
```

Remember that the CMyClass::Serialize() method serializes a WORD, DWORD, and a CPoint. Figure 5-1 shows the steps taken by MFC and your CDocument/CObject derivatives once the user has chosen a name for the file to which the document is written.

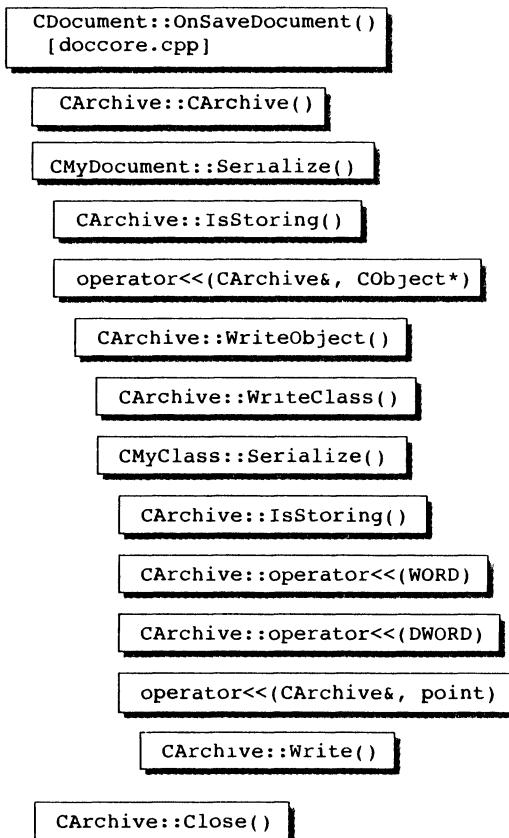


Figure 5-1. Steps taken during a serialization store operation

In step 1, CDocument::OnSaveDocument() creates a CArchive object in store mode and attaches it to a file already opened by the common dialog (pFile). Next, OnSaveDocument() calls the document's Serialize() method and finally closes the archive. Here's a look at the relevant code from CDocument::OnSaveDocument():

```

// ...
CArchive saveArchive(pFile, CArchive::store |
                     CArchive::bNoFlushonDelete);
saveArchive.m_pDocument = this;
TRY {
    Serialize(saveArchive);           // save me
  
```

```

saveArchive.Close();
ReleaseFile(pFile, FALSE);
}
// ...

```

In steps 3–5, the CMyDocument::Serialize() function is called, which checks if the archive is storing via IsStoring() and uses the insertion operator for writes (step 5).

In step 6–8, the insertion operator calls the CArcive::WriteObject() function, which calls WriteClass(), then Serialize, for the CMyDocument object.

Steps 9–13 write the actual data into the output buffer. Finally, the archive is closed by CDocument::OnSaveDocument() in step 14.

Let's look at a read operation (extraction) of the same object, as shown in Figure 5-2.

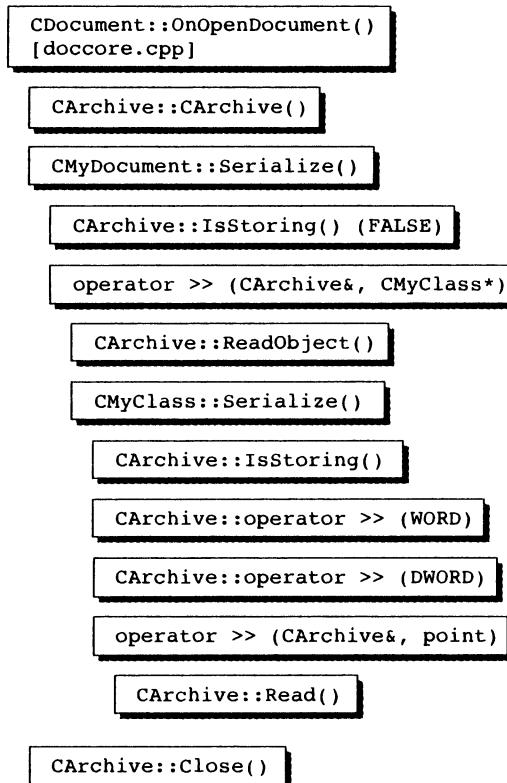


Figure 5-2. Steps in a serialization read operation

Basically, the framework “unwinds” what it wrote in the save operation when reading. Using this approach, the framework can make sure that data is read in the order that it was written.

Serialization Performance

You'll recall that when we went over the CArchive declaration, we briefly mentioned the maps and arrays used by MFC during serialization. Let's take a closer look at what these are used for.

Think about an application, like scribble, that has to write thousands of some type of object, for example, a CPoint. Isn't it wasteful to write out all of the common CPoint information thousands of times? MFC gets around this problem by serializing the CRuntime member of the first object type. For subsequent objects of the same type, MFC writes out a reference (for example, 1 meaning the first object) instead of complete CRuntime information. MFC stores the reference and object name in a map so that when other objects of the same type are written out, only the reference will be stored.

When reading, CArchive maintains an array of objects already read. When it reads a CRuntime reference instead of a full CRuntimeClass structure, CArchive looks up the reference in the array and uses the CRuntimeClass information from the array. In other words, CArchive keeps an array so that it can decode the references to CRuntimeClass information that was written out.

Another serialization performance consideration is the effects of serializing large archives of data. At greater than 16K of data, two APIs can help you tweak the performance of MFC serialization:

`SetLoadParams()` and `SetStoreParams()`. These values let you tweak the hash table size of the storing map and the grow-by size of the reading array to suit your needs. Review Chapter 4 if you don't remember how these values affect the MFC collections.

Serialization and Abstract Base Classes

If you try to use the serialization macros with an abstract base class, you will get an error message about the `CreateObject()` method when compiling. To get around this problem, you should define your own `IMPLEMENT_SERIAL` macro that looks something like this:

```
#define IMPLEMENT_SERIAL_ABC(class_name, base_class_name, wSchema) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
    NULL) \
CArchive& operator>>(CArchive& ar, class_name* &pOb) \
{ pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
return ar; }
```

The new macro eliminates the `CreateObject()` reference and lets you use the macros in an abstract base class.

A CRuntimeClass Status Update

Now that we've covered CObject serialization, we know much more about the members of the CRuntimeClass structure. Let's review CRuntimeClass to see which members we've left out.

- `LPCSTR m_lpszClassName`—This is the name of the class and is written out and read in as part of the class "state" during serialization.
- `UINT m_wSchema`—Also a class state. Gives the class version information.
- `void Store(CArchive& ar) const`—CArchive::WriteClass() calls this member function to write out the CRuntimeClass structure information about the class.
- `static CRuntimeClass * Load(CArchive& ar, UINT * pwSchemaNum);`—CArchive::ReadClass() calls Load() to read in the CRuntimeClass information written out by Store().

That leaves us with only two CRuntimeClass members that we haven't unearthed yet. (You never thought MFC internals would be like archaeology, did you?)

```
int m_nObjectSize;
CRuntimeClass * m_pNextClass;
```

Let's look at the last bit of functionality provided by CObject, diagnostic support, and discover how these last two members are used by MFC.

CObject Diagnostic Support

CObject diagnostic support comes in two flavors: basic diagnostics and advanced memory diagnostics, such as memory leak detection and memory statistics.

Before looking at how these features are implemented, let's quickly review how to use them.

Diagnostic Output

Diagnostic output is similar to old-fashioned "printf-debugging". MFC provides a printf-like macro called TRACE and also a slightly more sophisticated feature that lets you dump a C++ object's state at run time.

The TRACE Macro

The TRACE macro handles printf-like output. To turn on tracing, you have to run the MFC tracer utility. Once on, all output is sent to your Visual C++ debugger window. For example:

```
TRACE("At this point, my CPoint is: %d %d\n", mypoint.x, mypoint.y);
```

The Object Dump

All COBJECT-derived MFC classes have a Dump() member function that you can call to print out the state of a class. For example, here's how we might implement the dump routine for the CMYClass class:

```
#ifdef _DEBUG
void CMYClass::Dump(CDumpContext& dc)
{
    //Dump base class first
    CObject::Dump(dc);
    dc << "Type is: " << m_wType << "\n"
    << "Data is: " << m_dwData << "\n"
    << "Point is: " << m_ptMiddle.x << " "
    << m_ptMiddle.y << "\n";
}
#endif // end _DEBUG
```

Note that usually you place your Dump() routines in only the debug builds of your application.

Run-Time Checking

MFC provides some handy ways to check for certain error states at run time through assertions.

Assertions

The ASSERT macro is a debug-build macro that lets you check the state of something at run time and automatically display a warning message box if the specified test argument fails. If you've been doing much MFC programming, you're all too familiar with this macro. Figure 5-3 shows an example of a typical assertion.

To use the ASSERT macro, just pass in any Boolean test, function, and so on. If the argument result is false, the macro will assert. If it is true, the macro continues without taking action.

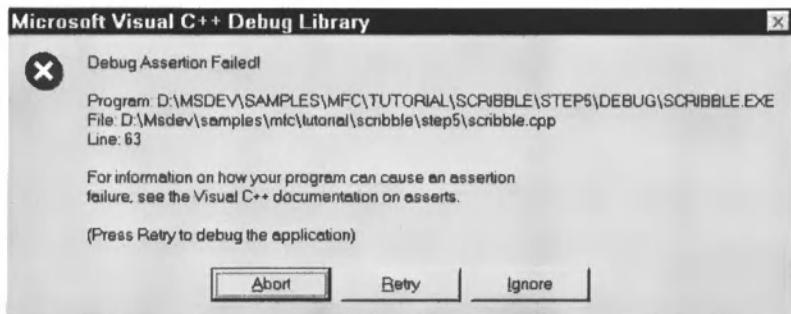


Figure 5-3. Everyone's friend, the ASSERT message box

Incidentally, MFC itself uses over 5000 ASSERT macros! You can tell Microsoft puts a lot of stake in this debugging tool. They use the heck out of it. This is good practice for you, as well. You should ASSERT the state of any of your internal variables anytime you assume anything. For example, you should always ASSERT a window handle is valid before using it.

Object-Validity Checking

In addition to the Dump() member function, most CObject derivatives implement an AssertValid() member function, in which the object checks its member's validity at run time. Any other object can call AssertValid() to make sure the object is in a safe state. If the object is not in a safe state, AssertValid() should trip an assertion to inform the developer as soon as possible.

Here's an example implementation of CMYClass::AssertValid():

```
#ifdef _DEBUG
void CMYClass::AssertValid()
{
    //Call inherited first
    CObject::AssertValid();
    ASSERT(m_wType > 0xff00);
    ASSERT(m_dwData != 0);
    ASSERT(m_ptMiddle.x != 0 && m_ptMiddle.y != 0);
}
#endif //End _DEBUG
```

MFC provides a convenient macro, ASSERT_VALID, that you can call on any CObject derivative. In addition to calling AssertValid(), this macro also does some extra run-time checking. For example, if we had a document with a CMYClass, we could use this statement in the document's AssertValid() member function:

```
ASSERT_VALID(m_myclass);
```

Memory Diagnostics

You may have seen some output after one of the MFC programs that you were debugging exited. It probably looked something like Listing 5-10.

Listing 5-10. Sample memory diagnostic output

```
Detected memory leaks!
Dumping objects ->
{100} a CDialog at $659328

m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{96} a CDialog at $658A10
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{92} a CDialog at $658454
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{88} a CDialog at $657B28
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{84} a CDialog at $657190
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

Object dump complete.
Process 0xFFFF35B63 terminated, exit code 0 (0x0).
```

This is the MFC memory-leak-detection feature at work. Though this is a pretty handy feature, MFC's advanced memory diagnostics are actually much more powerful than this simple usage. For example, you can use the memory diagnostics to

pinpoint leaks to the line, determine how much memory your program uses during its life, and even print statistics about memory use.

Detect a Memory Leak

To detect a memory leak, use the Checkpoint() member function of the CMemoryState class to wrap the area you are checking for leaks. Then you can use the Difference() member function to compare the two checkpoints and look for a leak between Checkpoint() calls.

Listing 5-11 shows an example using this technique:

Listing 5-11. How to detect a memory leak

```
//...
#ifndef _DEBUG
    CMemoryState startMemState, stopMemState, diffMemState;
    startMemState.Checkpoint(); //Start checking here
#endif //End _DEBUG
    CString abc = "Free me!";
//...
#ifndef _DEBUG
    stopMemState.Checkpoint();
    if (diffMemState.Difference(startMemState,stopMemState)) {
        TRACE("DANGER DANGER! Memory leak! ALERT!");
    }
#endif //End _DEBUG
```

Test Maximum Memory Usage

You can tell MFC to stop memory deallocations (frees and deletes) by adding the bitwise-OR delayFreeMemDF enumerated value into the afxMemDF global variable.

For example:

```
afxMemDF |= delayFreeMemDF;
```

By doing this, you can use the memory statistics feature to see how much memory your program is using.

Check Memory

If you want MFC to check all memory allocations and deallocations for you, OR in the value: checkAlwaysMemDF. For example:

```
afxMemDF |= checkAlwaysMemDF.
```

causes the framework to always check memory allocations and deallocations.

Review Memory Statistics

The CMemoryState class also has a `DumpStatistics()` member function that generates output like this:

```
0 bytes in 0 Free Blocks.
52 bytes in 1 Object Blocks.
0 bytes in 0 Non-Object Blocks.
Largest number used: 52 bytes.
Total allocations: 52 bytes.
```

Dump All Objects

The CMemoryState class even lets you dump the lines, addresses, and sizes of all leaked objects. That's pretty powerful. Call the `DumpAllObjectsSince()` member function to get output like Listing 5-12.

Listing 5-12. A dump of all objects

```
Dumping objects ->
{34} strcore.cpp(78) : non-object block at $00651BEO, 8 bytes long
{31} plex.cpp(28) : non-object block at $00651AD4, 132 bytes long
{30} a CObjfunView object at $00651A78, 52 bytes long
{28} a CMDIChildWnd object at $0065192C, 200 bytes long
{27} plex.cpp(28) : non-object block at $00651880, 132 bytes long
{26} a CObjfunDoc object at $006517F8, 96 bytes long
{25} array_p.cpp(111) : non-object block at $006517BC, 20 bytes long
{24} array_p.cpp(72) : non-object block at $00651790, 4 bytes long
{23} winfrm2.cpp(66) : a CDockBar object at $006516E4, 132 bytes long
{22} array_p.cpp(72) : non-object block at $006516B8, 4 bytes long
{21} winfrm2.cpp(66) : a CDockBar object at $0065160C, 132 bytes long
{20} array_p.cpp(72) : non-object block at $006515E0, 4 bytes long
{19} winfrm2.cpp(66) : a CDockBar object at $00651534, 132 bytes long
{17} winfrm2.cpp(66) : a CDockBar object at $0065145C, 132 bytes long
{16} bardock.cpp(491) : non-object block at $006513AC, 136 bytes long
{13} plex.cpp(28) : non-object block at $00650FB0, 132 bytes long
{12} strcore.cpp(78) : non-object block at $00650B40, 7 bytes long
{10} a CMainFrame object at $00650920, 456 bytes long
{9} plex.cpp(28) : non-object block at $00650874, 132 bytes long
{8} strcore.cpp(78) : non-object block at $00650818, 49 bytes long
{7} a CMultiDocTemplate object at $00650768, 136 bytes long
{5} strcore.cpp(78) : non-object block at $00650704, 7 bytes long
{4} strcore.cpp(78) : non-object block at $006506C8, 17 bytes long
{3} non-object block at $0065066C, 52 bytes long
{2} non-object block at $00650618, 44 bytes long
```

Inside COBJECT Diagnostic Support

Now that we've had that quick refresher on how to use the COBJECT diagnostic support, let's see how it's implemented inside MFC. As with serialization, anything this easy has to have a good deal of work going on behind the scenes.

Before we get started, a word on MFC versions. In MFC 4.0, the diagnostic support has been boosted to cover all mallocs and frees by moving much of the checking into the C Runtime library. In MFC 3.x and lower, only MFC objects were tracked by the diagnostic support. When appropriate, any version-specific explanations will be clearly marked.

First we'll look at how the simple diagnostics are implemented. Then we'll examine how MFC implements the nifty advanced memory diagnostic features.

Most of the macros and functions covered in this section are declared in the MFC file INCLUDE\AFX.H. The source code is in either the AFXMEM.CPP or AFTLS.CPP files.

Diagnostic Output

Let's start by looking at the declaration of the TRACE macro from AFX.H:

```
#define TRACE          ::AfxTrace
```

So this just causes the global AfxTrace() to be called. The AfxTrace() implementation lives in DUMPOUT.CPP, as shown in Listing 5-13.

Listing 5-13. AfxTrace() implementation from DUMPOUT.CPP

```
void AfxTrace(LPCTSTR lpszFormat, ...)
{
#ifdef _DEBUG // all AfxTrace output is controlled by afxTraceEnabled
    if (!afxTraceEnabled)
        return;
#endif
    va_list args;
    va_start(args, lpszFormat);
    int nBuf;
    TCHAR szBuffer[512];
    nBuf = _vstprintf(szBuffer, lpszFormat, args);
    ASSERT(nBuf < _countof(szBuffer));
    if ((afxTraceFlags & traceMultiApp) && (AfxGetApp() != NULL))
        afxDump << AfxGetApp()->m_pszExeName << ":" ;
    afxDump << szBuffer;
    va_end(args);
}
```

afxTraceEnabled is a Boolean that is declared in AFX.H and initialized in DUMPINIT.CPP in the constructor for the static object _AFX_DEBUG_STATE. Since this is a static object, its constructor will be called before the application starts running.

```
_AFX_DEBUG_STATE::_AFX_DEBUG_STATE()
{
    afxTraceEnabled =
        ::GetPrivateProfileInt(szDiagSection, szTraceEnabled,
        !afxData.bWin31,szIniFile);
    afxTraceFlags =
        ::GetPrivateProfileInt(szDiagSection, szTraceFlags,0, szIniFile);
    //...
}
```

At the top of DUMPINIT.CPP you will notice the following declarations for the GetPrivateProfileInt call:

```
static const TCHAR szIniFile[] = _T("AFX.INI");
static const TCHAR szDiagSection[] = _T("Diagnostics");
static const TCHAR szTraceEnabled[] = _T("TraceEnabled");
static const TCHAR szTraceFlags[] = _T("TraceFlags");
```

So the afxTraceEnabled and afxTraceFlags globals are just reading some values from the AFX.INI profile file. All the MFC tracer application does is toggle the values in this file.

Once TRACE determines that it should generate output by checking afxTraceEnabled, it goes through the variable list and outputs the results of vstprintf() (which provides all the printf functionality) to afxDump.

Let's take a look at afxDump and figure out how it handles all of the << operators and gets the output to your VC++ debugger window.

afxDump and CDumpContext

afxDump is declared in AFX.H as this:

```
#ifdef _DEBUG
    extern AFX_DATA CDumpContext afxDump;
#endif //End _DEBUG
```

The CDumpContext helper class is also declared in AFX.H. The CDumpContext declaration is in Listing 5-14.

Listing 5-14. The CDumpContext class declaration

```

class CDumpContext
{
public:
    CDumpContext(CFile* pFile = NULL);
// Attributes
    int GetDepth() const;           // 0 => this object, 1 => children objects
    void SetDepth(int nNewDepth);
// Operations
    CDumpContext& operator<<(LPCTSTR lpsz);
    CDumpContext& operator<<(LPCSTR lpsz); // automatically widened
    CDumpContext& operator<<(const void* lp);
    CDumpContext& operator<<(const CObject* pOb);
    CDumpContext& operator<<(const CObject& ob);
    CDumpContext& operator<<(BYTE by);
    CDumpContext& operator<<(WORD w);
    CDumpContext& operator<<(UINT u);
    CDumpContext& operator<<(LONG l);
    CDumpContext& operator<<(DWORD dw);
    CDumpContext& operator<<(float f);
    CDumpContext& operator<<(double d);
    CDumpContext& operator<<(int n);
    void HexDump(LPCTSTR lpszLine, BYTE* pby, int nBytes, int nWidth);
    void Flush();
// Implementation
protected:
    // dump context objects cannot be copied or assigned
    CDumpContext(const CDumpContext& dcSrc);
    void operator=(const CDumpContext& dcSrc);
    void OutputString(LPCTSTR lpsz);
    int m_nDepth;
public:
    CFile* m_pFile;
};

```

CDumpContext defines operators for the C++ and Windows types just like CArchive so that you can use the << operator with those types. Let's look at one of the CDumpContext operators to see how they are implemented and get a feel for what this class is doing. Here's the code for the WORD operator:

```

CDumpContext& CDumpContext::operator<<(WORD w)
{
    TCHAR szBuffer[32];
    wsprintf(szBuffer, _T("%u"), (UINT) w);
    OutputString(szBuffer);
    return *this;
}

```

All of the operators are similar: they call wsprintf() to change the value into a string representation. Then the operators pass the string to CDumpContext::OutputString(), which performs actual output. OutputString() is implemented in DUMPCONT.CPP, as shown in Listing 5-15.

Listing 5-15. Pseudocode for CDumpContext::OutputString() from DUMPCONT.CPP

```
void CDumpContext::OutputString(LPCTSTR lpsz)
{
#ifdef _DEBUG
    // all CDumpContext output is controlled by afxTraceEnabled
    if (!afxTraceEnabled)
        return;
#endif
    // use C-runtime/OutputDebugString when m_pFile is NULL
    if (m_pFile == NULL){
        AfxOutputDebugString(lpsz);
        return;
    }
    // otherwise, write the string to the file
    m_pFile->Write(lpsz, lstrlen(lpsz)*sizeof(TCHAR));
}
```

When the global afxDump is created, m_pFile is NULL (the default), so it is sending all of its output to the AfxOutputDebugString() function. If you create your own CDumpContext and give it a CFile pointer in the constructor, OutputString() prints the string to the file and you get a handy log of the output.

Version Note

In MFC 3.x and lower, CDumpContext..OutputString() calls the global OutputDebugString() instead of AfxOutputDebugString(). Source for OutputDebugString() is not provided, since it is a Windows API.

The declaration of AfxOutputDebugString() in AFX.H is actually a macro:

```
#ifdef _UNICODE
#define AfxOutputDebugString(lpsz) \
    do \
    { \
        int _convert; _convert = 0; \
        _RPT0(_CRT_WARN, W2CA(lpsz)); \
    } while (0)
#else
#define AfxOutputDebugString(lpsz) _RPT0(_CRT_WARN, lpsz)
#endif
```

AfxOutputDebugString() calls into the C Runtime library, so we have to assume that somewhere down in the bowels of that library Microsoft is connecting the CDumpContext output to their debug window.

Object Dumping

Now that we understand what the TRACE macro is doing, how it uses the CDumpContext helper class, and how it is initialized from profile strings, let's look at the mechanism behind object dumping.

Take another look at the declaration for CObject in Listing 5-1. Notice that CObject has a virtual Dump() function you can override. Let's look at the implementation of CObject::Dump() from OBJCORE.CPP:

```
#ifdef _DEBUG
void CObject::Dump(CDumpContext& dc) const
{
    dc << "a " << GetRuntimeClass()->m_lpszClassName << " at " <<
        (void*)this << "\n";
}
#endif // _DEBUG
```

The CObject::Dump() is just dumping the class name (stored in CRuntimeClass) and the address of the “this” pointer.

Dump() output also uses the CDumpContext operators that the TRACE macro uses.

AssertValid() and Assertions

Let's look at how MFC handles assertions. Specifically, we'll see how the AssertValid() run-time object-checking capabilities are implemented.

The declaration for the ASSERT macro is in AFX.H:

```
#define ASSERT(f) \
    do \
    { \
        if (!(f) && AfxAssertFailedLine(THIS_FILE, __LINE__)) \
            AfxDebugBreak(); \
    } while (0)
```

The AfxAssertFailedLine() global function lives in OBJCORE.CPP. It first blocks any active threads and then displays the familiar message box with the file name and line number.

Now that you see what ASSERT really does, you may notice that you can set a breakpoint at AfxAssertFailedLine() before starting the debugger. Since it allows

you to get the assertion reported as a breakpoint in the debugger before the message box is brought up, setting a breakpoint here makes debugging some tricky situations like WM_ACTIVATE handling much easier—and some multithreaded situations too.

You may have seen this declaration at the top of MFC source files:

```
#ifdef _DEBUG
#undef THIS_FILE
    static char THIS_FILE[] = __FILE__;
#endif
```

This creates the space for the current file name and places the file name in. The compiler automatically substitutes __FILE__ and __LINE__ for you.

When the ASSERT macro calls AfxDebugBreak(), control turns over to the Visual C++ debugger, so you can interactively debug assertions.

AssertValid()

Just like Dump(), AssertValid() is declared as virtual in CObject. The implementation of CObject::AssertValid() can be found in OBJCORE.CPP. It simply verifies that the “this” pointer is not NULL.

```
void CObject::AssertValid() const
{
    ASSERT(this != NULL);
}
```

Let’s look at the ASSERT_VALID macro and see what else it’s doing. ASSERT_VALID is defined as the following:

```
#define ASSERT_VALID(pOb)  (::AfxAssertValidObject(pOb, THIS_FILE, __LINE__))
```

The AfxAssertValidObject() global function lives in OBJCORE.CPP. Listing 5-16 shows the pseudocode for AfxAssertValidObject():

Listing 5-16. Pseudocode for AfxAssertValidObject()

```
void AfxAssertValidObject(const CObject* pOb, LPCSTR lpszFileName, int nLine)
{
    if (pOb == NULL) {
        TRACE0("ASSERT_VALID fails with NULL pointer.\n");
        if (AfxAssertFailedLine(lpszFileName, nLine))
            AfxDebugBreak();
        return;      // quick escape
    }
}
```

```

if (!AfxIsValidAddress(pOb, sizeof(CObject))) {
    TRACE0("ASSERT_VALID fails with illegal pointer.\n");
    if (AfxAssertFailedLine(lpszFileName, nLine))
        AfxDebugBreak();
    return;      // quick escape
}
// check to make sure the VTable pointer is valid
ASSERT(sizeof(CObject) == sizeof(void*));
if (!AfxIsValidAddress(*(void**)pOb, sizeof(void*), FALSE)) {
    TRACE0("ASSERT_VALID fails with illegal vtable pointer.\n");
    if (AfxAssertFailedLine(lpszFileName, nLine))
        AfxDebugBreak();
    return;      // quick escape
}
if (!AfxIsValidAddress(pOb, pOb->GetRuntimeClass()->m_nObjectSize)) {
    TRACE0("ASSERT_VALID fails with illegal pointer.\n");
    if (AfxAssertFailedLine(lpszFileName, nLine))
        AfxDebugBreak();
    return;      // quick escape
}
pOb->AssertValid();
}

```

First, `AfxAssertValidObject()` checks that the object isn't NULL. Next, `AfxAssertValidObject()` makes sure the object is the correct size and then checks the validity of its address. Finally, `AfxAssertValidObject()` verifies that the size of the object matches the information stored in `CRuntimeClass::m_nObjectSize` and calls the object's `AssertValid()` member function. Your object may have specific state information. Because `AssertValid()` is virtual, you may override it to assert the conditions specific to your object.

Though this is obviously an expensive macro, it is great to know and use when you are in tricky debug situations. If an object passes through this gauntlet of tests, you can be fairly sure the problem is not the object itself.

In `ASSERT_VALID`, `AfxIsValidAddress()` seems key to the MFC memory checking. Let's take a quick look at it so that you can use it in your own MFC programs.

AfxIsValidAddress()

This helper function is declared in `AFX.H` as follows:

```
BOOL AfxIsValidAddress(const void* lp, UINT nBytes, BOOL bReadWrite = TRUE);
```

It is implemented in `VALIDADD.CPP` as this:

```
BOOL AfxIsValidAddress(const void* lp, UINT nBytes, BOOL bReadWrite)
{
```

```

    return (lp != NULL && !IsBadReadPtr(lp, nBytes) &&
           (!bReadWrite || !IsBadWritePtr((LPVOID)lp, nBytes)));
}

```

This function uses basic Win32 APIs (`IsBadReadPter()` and `IsBadWritePter()`) to check the pointer. It uses the `bReadWrite` argument (default is TRUE) to check if the pointer is writable.

If you look in the MFC source code, you will find over 400 calls to `AfxIsValidAddress()`, so this is a very important utility function. Be sure to use it liberally in your own MFC applications.

Usually, MFC wraps `AfxIsValidAddress()` in an `ASSERT` macro such as this one:

```
ASSERT(AfxIsValidAddress(lpLogFont, sizeof(LOGFONT), FALSE));
```

It is a good habit to use `AfxIsValidAddress()` to test every pointer passed into your member functions to make sure that the developer isn't passing a garbage pointer. As with the `ASSERT` macros, using `AfxIsValidAddress()` slows down your debug build, but these macros are all compiled away in release builds.

That wraps up our coverage of the simple output diagnostics. Let's move on and see how MFC implements the advanced memory diagnostics, such as memory leak detection.

Advanced Memory Diagnostics

Version Note

The memory diagnostic implementation varies a great deal between MFC 3.x and 4.x. As previously noted, version 4.x uses mostly C Runtime library routines, where MFC 3.x rolled its own support.

In this section, we'll take a look at both MFC 3.x and 4.x to help you understand what is going on in the C Runtime library without having to get our hands dirty in that large body of code.

CMemoryState

When using the advanced diagnostics, you probably noticed another helper class called `CMemoryState`. This helper class is largely undocumented. Let's take a look at the declaration in Listing 5-17 and some of the key member functions to see how this helper class works.

Listing 5-17. The CMemoryState helper declaration

```

struct CMemoryState
{
// Attributes
    enum blockUsage {
        freeBlock,      // memory not used
        objectBlock,   // contains a CObject derived class object
        bitBlock,       // contains ::operator new data
        crtBlock,       // New in MFC 4.
        ignoredBlock,  // New in MFC 4.
        nBlockUseMax  // total number of usages
    };
    _CrtMemState m_memState;
    //Note: MFC 3.x had: CBlockHeader * m_pBlockHeader; instead
    LONG m_lCounts[nBlockUseMax];
    LONG m_lSizes[nBlockUseMax];
    LONG m_lHighWaterCount;
    LONG m_lTotalCount;
    CMemoryState();
// Operations
    void Checkpoint(); // fill with current state
    BOOL Difference(const CMemoryState& oldState,
                    const CMemoryState& newState); // fill with difference
    void UpdateData();
// Output to afxDump
    void DumpStatistics() const;
    void DumpAllObjectsSince() const;
};


```

Version Note

Because MFC 4.0 pushes most of the CMemoryState functionality into the C Runtime library, this explanation of CMemoryState is based on the MFC 3.x code, which is much easier to understand.

Each block of allocated memory tracked by CMemoryState falls into one of these categories:

- freeBlock—Blocks whose freeing has been delayed by using delayFreeMemDF.
- objectBlock—The number of CObject objects that remain or have been leaked.
- bitBlock—The number of non-CObject objects allocated.
- crtBlock (MFC 4.0 only)—The number of C run-time blocks allocated.
- ignoredBlock (MFC 4.0 only)—The number of blocks ignored by the MFC checking.

CMemoryState maintains both the counts and the sizes of these types of allocations in the m_lCounts and m_lSizes member arrays.

The m_lHighWaterCount member stores the largest allocation, and m_lTotalCount tracks the total number of allocations. (What illustrious names!)

The m_memState or m_pBlockHeader member variables point to the head of a linked list of memory blocks.

Now that we know what CMemoryState stores, the big question is How does CMemoryState know about all of the object allocations? MFC tracks allocations and deallocations through the ability in C++ to override the new operator.

The Debug New Operator

You may have noticed these lines in any source file generated by ClassWizard:

```
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
```

Defining _DEBUG redirects all of your “new” operations to call DEBUG_NEW instead.

Here's how DEBUG_NEW is defined in AFX.H:

```
#define DEBUG_NEW new(THIS_FILE, __LINE__)
```

This macro causes debug builds to use a different new operator, which tracks memory allocations and stores file names and line numbers.

Here's the declaration for the overriden global debug new operator:

```
void* operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
```

From Listing 5-1, You may also remember that CObject defines several new and delete operators. To understand why there are two different debug new operators (one global and the other for CObject), let's look at the implementation of both from AFXMEM.CPP. Listing 5-18 contains the pseudocode for both debug new operators.

Listing 5-18. The two debug new operators (can you find the difference?)

```
void* operator new(size_t nSize, LPCSTR lpszFileName, int nLine)
{
    // memory corrupt before global new
    if (afxMemDF & checkAlwaysMemDF)
        ASSERT(AfxCheckMemory());
```

```

void* p = AfxAllocMemoryDebug(nSize, FALSE, lpszFileName, nLine);
if (p == NULL) {
    TRACE1("::operator new(%u) failed - throwing exception.\n", nSize);
    AfxThrowMemoryException();
}
return p;
}
void* CObject::operator new(size_t nSize, LPCSTR lpszFileName, int nLine)
{
    // memory corrupt before 'CObject::new'
    if (afxMemDF & checkAlwaysMemDF)
        ASSERT(AfxCheckMemory());
    void* p = AfxAllocMemoryDebug(nSize, TRUE, lpszFileName, nLine);
    if (p == NULL) {
        TRACE1("CObject::operator new(%u) failed - throwing exception.\n",
               nSize);
        AfxThrowMemoryException();
    }
    return p;
}

```

There is only one slight difference between the two operators. Can you spot it? Here are three hints:

1. Look at the call to AfxAllocMemoryDebug().
2. Look at the second argument.
3. OK, OK, one has TRUE and the other has FALSE as the argument. We'll see why momentarily.

Now that we know there's a new operator specifically for debug builds, let's look at how the operator stores the memory allocation information. We'll do this by examining the global helper function AfxAllocMemoryDebug().

Listing 5-19 has the pseudocode for this helper.

Listing 5-19. Pseudocode for AfxAllocMemoryDebug()

```

void* AfxAllocMemoryDebug(size_t nSize, BOOL bIsObject, LPCSTR
lpszFileName, int nLine)
{
    if (nSize == 0)
        TRACE("Warning: Allocating zero length memory block.\n");
    if (nSize > (size_t)SIZE_T_MAX - nNoMansLandSize -
        sizeof(CBlockHeader)) {
        TRACE1("Error: memory allocation: tried to allocate %u bytes --\n",
               nSize);
        TRACE0("\tobject too large or negative size.\n");
    }
}

```

```

AfxThrowMemoryException();
}
// keep track of total amount of memory allocated
lTotalAlloc += nSize; lCurAlloc += nSize;
if (lCurAlloc > lMaxAlloc)
    lMaxAlloc = lCurAlloc;
struct CBlockHeader* p =
    (struct CBlockHeader*) malloc(sizeof(CBlockHeader) + nSize +
        nNoMansLandSize);
if (p == NULL)
    return NULL;
LockDebugMemory(); // block other threads
if (pFirstBlock)
    pFirstBlock->pBlockHeaderPrev = p;
p->pBlockHeaderNext = pFirstBlock;
p->pBlockHeaderPrev = NULL;
p->lpszFileName = lpszFileName;
p->nLine = nLine;
p->nDataSize = nSize;
p->use = bIsObject ? CMemoryState::objectBlock : CMemoryState::bitBlock;
p->lRequest = lRequest;
// fill in gap before and after real block
memset(p->gap, bNoMansLandFill, nNoMansLandSize);
memset(p->pbData() + nSize, bNoMansLandFill, nNoMansLandSize);
// fill data with silly value (but non-zero)
memset(p->pbData(), bCleanLandFill, nSize);
// link blocks together
pFirstBlock = p;
UnlockDebugMemory(); // release other threads
return (void*)p->pbData();
}

```

This fairly complex function is the key to understanding the way MFC tracks memory, so let's dissect AfxAllocMemoryDebug() in steps.

1. AfxAllocMemoryDebug() first verifies that the size is neither negative nor too large.
2. Next, AfxAllocMemoryDebug() updates the total allocation count and max count if it is larger than a previously allocated block.
3. AfxAllocMemoryDebug() then allocates the memory. Notice the size used in the call to malloc():

```
malloc(sizeof(CBlockHeader) + nSize + nNoMansLandSize);
```

MFC allocates more space than necessary and prefaces the memory block with a CBlockHeader structure so that it can keep track of the memory. In the CBlockHeader structure the last item is a gap of bytes. MFC also leaves space

after the data, called a “no man’s land.” MFC uses the space on either side of the actual data in the memory to write an uncommon memory pattern, or sentinel. By checking this sentinel, MFC can quickly tell if a program has under- or over-written the allocated memory and alert the user about the problem.

Figure 5-4 shows graphically how MFC allocates the memory.

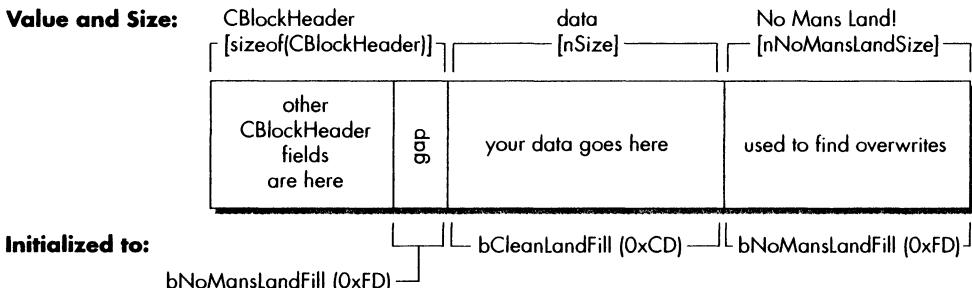


Figure 5-4. How MFC allocates memory in debug mode to detect clobbers

4. After allocating the memory, AfxAllocMemoryDebug() locks all of the other threads.
5. Next, it adds the current memory block to the linked list of blocks (pFirstBlock->pBlockHeaderPrev = p;).
6. It then puts all of the information about the memory into the CBlockHeader, including file name, line number, size, and whether it’s an object or bit allocation based on the bIsObject argument (look at the global and CObject debug new operators again closely).
7. Next, you will see two memset() calls that copy the sentinel values mentioned in step 3 before and after the memory block.
8. The next memset() call fills the allocated memory with nonzero values so that you can verify that your program is not making assumptions about zero-initialized memory.
9. Finally, the function returns a pointer to the memory just beyond the CBlockStructure so that your program is none the wiser that all this under-the-hood trickery is going on.

Back to CMemoryState

Now that we understand how MFC is storing the memory allocations, let’s look at how the key member functions of CMemoryState are implemented.

First, let's look at `CMemoryState::Checkpoint()`, from `AFXMEM.CPP`, in Listing 5-20.

Listing 5-20. The implementation of CMemoryState::Checkpoint(), from AFXMEM.CPP

```

void CMemoryState::Checkpoint()
{
    if (!(afxMemDF & allocMemDF))
        return; // can't do anything
    LockDebugMemory(); // block other threads
    m_pBlockHeader = pFirstBlock;
    for (int use = 0; use < CMemoryState::nBlockUseMax; use++)
        m_lCounts[use] = m_lSizes[use] = 0;
    struct CBlockHeader* p;
    for (p = pFirstBlock; p != NULL; p = p->pBlockHeaderNext) {
        if (p->lRequest == lNotTracked)
            // ignore it for statistics
        else if (p->use >= 0 && p->use < CMemoryState::nBlockUseMax){
            m_lCounts[p->use]++;
            m_lSizes[p->use] += p->nDataSize;
        }
        else{
            TRACE1("Bad memory block found at $%08lx.\n", (BYTE*)p);
        }
    }
    m_lHighWaterCount = lMaxAlloc;
    m_lTotalCount = lTotalAlloc;
    UnlockDebugMemory(); // release other threads
}

```

`CMemoryState::Checkpoint()` first initializes the `m_lCounts` and `m_lSizes` counter arrays to zero. Then `CheckPoint()` iterates through the linked list of memory blocks and adds the memory size and counts to the appropriate category in the `m_lCounts` and `m_lSizes` arrays. Finally, `Checkpoint()` updates the `m_lHighWaterCount` and `m_lTotalCount` members based on the last values set in the global `lMaxAlloc`/`lTotalAlloc`, which are set by `AfxAllocMemoryDebug()` if new maximums are hit.

Let's look at `CMemoryState::Difference()` to see how it uses the information stored in two checkpointed `CMemoryState` objects to determine if memory has leaked. Listing 5-21 shows the pseudocode.

Listing 5-21. The implementation of CMemoryState::Difference()

```

BOOL bSignificantDifference = FALSE;
for (int use = 0; use < CMemoryState::nBlockUseMax; use++){
    m_lSizes[use] = newState.m_lSizes[use] - oldState.m_lSizes[use];
    m_lCounts[use] = newState.m_lCounts[use] - oldState.m_lCounts[use];
    if ((m_lSizes[use] != 0 || m_lCounts[use] != 0) &&
        use != CMemoryState::freeBlock)
        bSignificantDifference = TRUE;
}
m_lHighWaterCount = newState.m_lHighWaterCount -
    oldState.m_lHighWaterCount;
m_lTotalCount      = newState.m_lTotalCount - oldState.m_lTotalCount;
return bSignificantDifference;
}

```

For each memory category, Difference() subtracts the accumulated values from each CMemoryState argument into its own m_lSizes and m_lCounts arrays. If the difference is nonzero and the category is not CMemoryState::freeBlock, a significant difference is flagged and TRUE is returned to the caller to indicate that a problem has been detected.

Now that you're starting to get a feel for CMemoryState, the implementation of DumpStatistics() in Listing 5-22 shouldn't surprise you.

Listing 5-22. The implementation of CMemoryState::DumpStatistics()

```

void CMemoryState::DumpStatistics() const
{
    for (int use = 0; use < CMemoryState::nBlockUseMax; use++){
        TRACE3("%ld bytes in %ld %hs Blocks.\n", m_lSizes[use],
               m_lCounts[use], blockUserName[use]);
    }
    TRACE1("Largest number used: %ld bytes.\n", m_lHighWaterCount);
    TRACE1("Total allocations: %ld bytes.\n", m_lTotalCount);
}

```

CMemoryState::DumpAllObjectsSince() iterates through the linked list of CBlockHeader structures checking all of the memory addresses and generating output specific to the type of object. This information includes the file name and line number of the allocation.

AfxCheckMemory()

You may be wondering where MFC checks those sentinels put in place by AfxAllocMemoryDebug(). The AfxCheckMemory() global helper function checks the sentinels, calling TRACE for every problem that it finds. If you are debugging and you suspect there is some memory being overwritten, call the AfxCheckMemory() function to quickly find where the problem is happening.

AfxDumpMemoryLeaks()

You may also be wondering where and how MFC generates a list of any memory leaks when exiting. In MFC 4.x, this is all handled by the C run-time exit module, which calls `_CrtDumpMemoryLeaks()` automatically on exit.

In MFC 3.x, this happens in WINMAIN.CPP after `CWinApp::Run()` by calling the global helper `AfxDumpMemoryLeaks()`. Here's a quick look at the MFC 3.x `AfxDumpMemoryLeaks()` function, in Listing 5-23.

**Listing 5-23. AfxDumpMemoryLeaks() from MFC 3.x, equivalent to
_CrtDumpMemoryLeaks() in MFC 4.x**

```
BOOL AfxDumpMemoryLeaks()
{
    // only dump leaks when there are in fact leaks
    CMemoryState msNow;
    msNow.Checkpoint();
    if (msNow.m_lCounts[CMemoryState::objectBlock] != 0 ||
        msNow.m_lCounts[CMemoryState::bitBlock] != 0){
        // dump objects since empty state since difference detected.
        TRACE0("Detected memory leaks!\n");
        afxDump.SetDepth(1);      // just 1 line each
        CMemoryState msEmpty;    // construct empty memory state object
        msEmpty.DumpAllObjectsSince();
        return TRUE;
    }
    return FALSE;    // no leaked objects
}
```

`AfxDumpMemoryLeaks()` just calls `_CrtDumpMemoryLeaks()` in MFC 4.x. However, you can still call it at any time to see what has leaked since the program started.

Cool AFX Helper Functions

Two interesting global AFX helper functions are available for use with debugging: `AfxDoForAllObjects()` and `AfxDoForAllClasses()`.

AfxDoForAllObjects()

`AfxDoForAllObjects()` takes a pointer to a function and a `void*` pass-through pointer. `AfxDoForAllObjects()` iterates through the linked list of memory objects and passes them to the user-provided function as `CObject` pointers along with the pass-through argument. In MFC 3.x, this function iterates through the `CBlockHeader` linked list. In MFC 4.x, it is implemented entirely in C Runtime routines. This function lives in `AFXMEM.CPP`.

AfxDoForAllClasses()

`AfxDoForAllClasses()` is similar to the `AfxDoForAllObjects()` routine, but it iterates through all of the `CRuntimeClass` structures and lets you pass them to a function, as we can see in its implementation:

```
void AfxDoForAllClasses(void (*pfn)(const CRuntimeClass*, void*),
                      void* pContext)
{
    // just walk through the simple list of registered classes
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    AfxLockGlobals();
    for (CRuntimeClass* pClass = pModuleState->m_classList; pClass != NULL;
         pClass = pClass->m_pNextClass)
        (*pfn)(pClass, pContext);
}
```

Aha! The last `CRuntimeClass` structure member, `m_pNextClass`, is finally revealed. MFC stores the class in this list via the `AFX_CLASSINIT` structure constructor generated by the `DECLARE/IMPLEMENT` macros covered earlier.

Putting It All Together

Now that we've covered all of `CObject`'s features, you should be able to flip back to the first listing of the `CObject` declaration and understand why `CObject` is set up that way and what the various members are doing.

To help tie all of the features together, let's quickly review what each data member in the `CRuntimeClass` structure is used for:

- `m_lpszClassName`—Used in serialization to store the state of the class.
- `m_nObjectSize`—Used by the debug utility function `AfxAssertValidObject()` to verify that the size of the class matches its `CRuntimeClass` equivalent. This could get confused during serialization or clobbered by a memory problem.
- `m_wSchema`—Serialization uses this WORD to give a class a version. If serialization is not supported, the `m_wSchema` is 0xffff. You can specify your own schema through the `IMPLEMENT_SERIAL` macro as the third argument. Or change it yourself at run time, now that you know about it and how to get to it.
- `m_pfnCreateObject`—Used for dynamic creation. The `DECLARE/IMPLEMENT_DYNCREATE` macros set this member function pointer to the `CMyClass::CreateObject()` member function. When you call

`CRuntimeClass::CreateObject()`, it calls `CMyClass::CreateObject()` through `m_pfnCreateObject`. `CMyClass::CreateObject()` just returns a new `CMyClass`.

- `m_pBaseClass`—Holds a pointer to the `CRuntimeClass` structure for the base class from which the current class is derived. Used by `IsKindOf()` and `IsDerivedFrom()` to determine the “polymorphic type” of the object.
- `m_pNextClass`—Maintains a simple linked list of all the classes that have been created.

Putting It into Practice

The information we have exposed about the `CObject` features is interesting, but is it useful? The answer is a resounding yes. It's very useful when you are in the debugger: you can now feel comfortable printing out and understanding the values of the previously undocumented helper class members. You can also use the information revealed in this chapter in other ways.

For example, we have implemented a debug dialog that displays a list of all the classes in the `CRuntimeClass` list and also all of the objects that are currently created. Figure 5-5 shows the results of the class list for a standard document/view MDI application. Figure 5-6 shows the objects for the same application.

Class Name	Size	Schema	BaseClass
CDialog	68	##	CWnd
CObjView	52	##	CView
CMDChildWnd	200	##	CFrameWnd
CObjDoc	96	##	CDocument
CDockBar	132	##	CCControlBar
CDockBar	132	##	CCControlBar
CDockBar	132	##	CCControlBar
CDockBar	132	##	CCControlBar
CMainFrame	456	##	CMDIFrameWnd
CMultiDocTemplate	136	##	CDocTemplate
CDynLinkLibrary	52	##	CCmdTarget

Figure 5-5. An example of accessing the `CRuntimeClass` list in your own applications

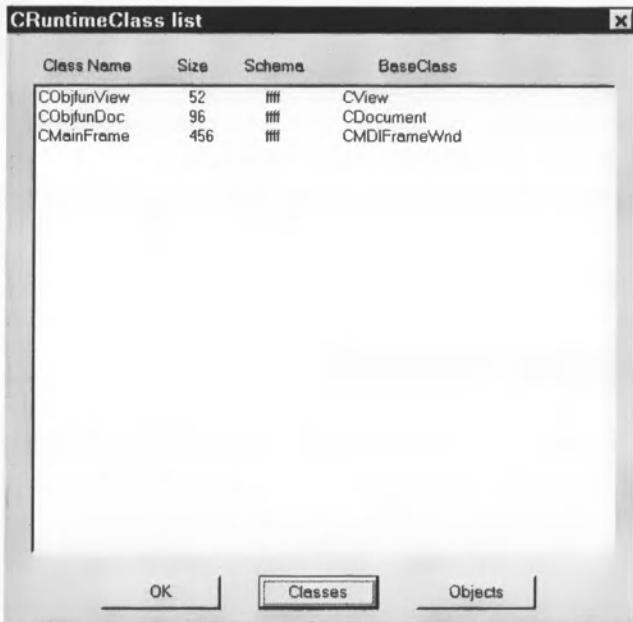


Figure 5-6. A snapshot of all the objects created at a moment in run time

The code for this sample is provided on the companion diskette in the chap05 directory. Feel free to investigate how the sample works and to extend it if desired.

Is It Worth It?

Now that we fully understand what we gain from deriving from `CObject`, let's revisit the question posed earlier in the chapter: "Does the MFC architecture with `CObject` as root cause performance problems?"

`C++` has to index the class virtual table (the table of virtual functions) at run time to determine the correct function to execute. Looking back, you will see that `CObject` has five virtual functions, two of which are in debug builds only, so they don't really affect release-build performance. The largest overhead with using virtual functions is the increased instance size from a virtual table. If you plan on using virtual functions, then `CObject` doesn't add any additional overhead.

Therefore, the trade-off in deriving from `CObject` is three extra virtual functions added to your classes' virtual table in exchange for RTTI, dynamic creation, serialization, and memory diagnostics. The choice is yours, but if there's even the

remote chance that your class will need any of the functionality provided by CObject, you should probably go for it. MFC itself has only a few classes that are not CObject-derived (which were covered in Chapter 4). Most of these are extremely small and don't have any virtual functions to begin with.

Conclusion

At this point in the book you should be able to open up the non-GUI MFC header file, AFX.H, and understand what every class, macro, and helper function is doing in there.

Why don't you try it right now to help you refresh what you've learned and prepare for the next chapters. We'll be covering how MFC implements dialogs, controls, document/views, and even advanced user-interface classes like splitter windows.

MFC Dialog and Control Classes

In this chapter we'll cover the internals of the MFC dialog and control classes. Because all these classes derive from `CObject`, `CCmdTarget`, and `CWnd`, make sure you have a good grasp of the MFC internals introduced in the previous chapters before you take on the MFC dialogs and controls.

First, we'll cover how MFC implements Windows dialogs and see how MFC dynamic data exchange and validation (a.k.a. DDX/DDV) work under the hood. Next, we'll investigate how MFC wraps the Windows common dialogs and the OLE dialogs. After common dialogs, we look at MFC property sheets, or tabbed dialogs.

Once we've finished with the MFC internals for the dialog classes, we'll shift gears in the last part of the chapter and examine the MFC control classes.

Finally, we'll see how Microsoft has beefed up MFC over the last several releases to provide support for the new and exciting Windows common controls.

CDialog: Modal and Modeless MFC Dialogs

The MFC class `CDialog` supports both modal and modeless dialogs. In this section we'll briefly review how to use `CDialog` and then we'll see how MFC implements the class. We'll also see how MFC initializes, exchanges, and validates dialog data.

Using `CDialog`

To use class `CDialog`, you create a `CDialog` derivative (usually using Class Wizard), which specializes `CDialog` to display the dialog you are after. Usually you tie your `CDialog` derivative to a dialog resource through an integer dialog ID or a resource dialog name. Figure 6-1 shows how to do this using Class Wizard.

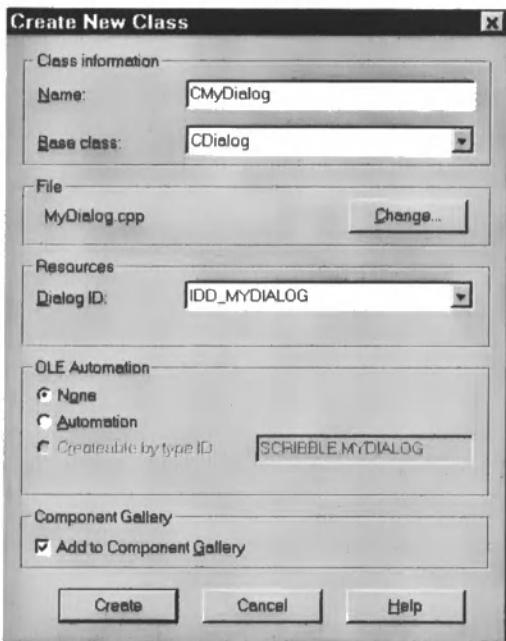


Figure 6-1. Class Wizard in action

Class CDIALOG supports both modal and modeless dialog creation and operations. Modal dialogs “freeze” the rest of the application and force the user to perform some operation before it will go away. A good example of a modal dialog box is the Common Color dialog box. Once the dialog box appears on the screen, the only way you can get rid of it is by pressing the OK button. Modeless dialogs, on the other hand, let the user continue using the application while it is displayed. Usually the user has to get rid of a modeless dialog manually. A good example of a modeless dialog box is the Spell Check dialog box in Word for Windows. Let’s review how to use both modal and modeless CDIALOGs in MFC.

Historical Note

Way back in MFC 1.0, there were two dialog classes, CDIALOG and CModalDialog. In post-1.0 versions, MFC merged the two classes into CDIALOG. For backward compatibility, even today, MFC has the following #define

```
#define CModalDialog CDIALOG
```

in AFXWIN.H, so that any MFC 1.0 applications that use CModalDialog will be mapped to CDIALOG and work properly.

Modal Dialogs

To create a modal CDIALOG dialog, you create a local object of your CDIALOG derivative and call the DoModal() member function. DoModal() initializes, creates, displays, and destroys the dialog for you.

After DoModal() returns, you can check its return value to find out if the user selected “OK” or “Cancel”. You can also access information stored in your CDIALOG derivative after DoModal() returns. Here’s an example of a typical MFC modal dialog:

```
void CMYView::DisplayOrderDialog()
{
    CMyDialog myDialog(ID_DLG_MYDIALOG);
    int nRetVal = myDialog.DoModal();
    if (nRetVal == IDOK) {
        //Do OK processing, maybe retrieve a value from myDialog?
    }
    else {
        //Do Cancel processing, maybe reset some values?
    }
}
```

In this example, ID_DLG_MYDIALOG, which was passed to the CMyDialog constructor, refers to a dialog template resource that lives in the application’s resources. CDIALOG provides constructors that take both the integer ID and the name of the dialog template for your convenience.

Modeless Dialogs

Modeless dialogs usually live longer than one function call, so they are created from the heap using the new operator. To be able to keep track of the dialog, most applications will need to store away a pointer to the modeless dialog in a member function so that it can be destroyed at some point (for example, at application exit, or when the user closes the dialog through a menu item).

After constructing a modeless CDIALOG, you have to call explicitly the Create() member function to display the dialog. Modeless dialogs are destroyed through the inherited CWnd::DestroyWindow() member function.

The following code snippet shows how to create, display, and destroy a modeless dialog.

```
//Construct it, store in a member variable.
m_pDlgMyDlgPtr = new CMyDialog;
//Display it, pass in the template reference
m_pDlgMyDlgPtr->Create(ID_DLG_MYDIALOG);
//Sometime later, nuke it.
```

```
m_pDlgMyDlgPtr->DestroyWindow();
m_pDlgMyDlgPtr = NULL;
```

Dialog Operations

In addition to modal and modeless creation, class CDialog supports numerous dialog operations. For example, by calling NextDlgCtrl(), PrevDlgCtrl(), or GotoDlgCtrl(), you can iterate through the controls in the dialog. There are also operations that let you set the help IDs and default IDs of the buttons in the dialog.

CDialog also provides overrides for changing the behavior of the OK and Cancel buttons and perform dialog initialization.

Certain derivatives for CDialog provide overrides specific to their functionality. For example, CFontDlg has overrides to specify the font.

CDialog is pretty easy to use once you get the hang of creating the dialog templates and whipping out a CDialog derivative with Class Wizard. Let's take a look at how MFC implements CDialog and see how much trouble MFC is going through for our benefit.

CDialog Internals

There are only a handful of Win32 APIs for creating dialogs:

- CreateDialog()—Creates a modeless dialog from a template resource.
- CreateDialogIndirect()—Creates a modeless dialog from a template pointer.
- DialogBox()—Creates a modal dialog from a template resource.
- DialogBoxIndirect()—Creates a modal dialog from a template pointer.

CDialog only calls the CreateDialogIndirect() API and implements modality itself instead of calling one of the DialogBox routines. MFC calls the “Indirect” version of the API, so that it can add extra checking to the resource loading process of the dialog template.

In the early days of MFC, calling CreateDialogIndirect() was pretty much all that CDialog did. In the later versions, new features have been added, and CDialog has gotten much more complex. For example, the addition of OLE controls has almost doubled the amount of code needed for CDialog, because a number of OLE interfaces are required for OLE control support.

We cover the OLE control exceptions in Chapter 15 and concentrate on how CDialog wraps and enhances the Win32 dialog APIs.

Class CDialog is declared in the AFXWIN.H header file and is implemented in the DLGCORE.CPP MFC source file. Some of the CDialog members are in the MFC inline file AFXWIN2.INL.

CDialog's Abbreviated Declaration

To see how CDialog works, first let's look at its abbreviated declaration, as shown in Listing 6-1. For brevity, the documented CDIALOG members have been omitted so that we can focus on the undocumented implementation details of the class.

Listing 6-1. The abbreviated CDIALOG declaration, from AFXWIN.H

```
class CDIALOG : public CWnd
{
    DECLARE_DYNAMIC(CDIALOG)
// Construction **omitted
// Attributes  **omitted
// Operations   **omitted
// Overridables **omitted
// Implementation **The good stuff!!
public:
    virtual ~CDIALOG();
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo);
    virtual BOOL CheckAutoCenter();
protected:
    UINT m_nIDHelp;           // Help ID (0 for none, see HID_BASE_RESOURCE)
    // parameters for 'DoModal'
    LPCTSTR m_lpszTemplateName; // name or MAKEINTRESOURCE
    HGLOBAL m_hDialogTemplate; // indirect (m_lpDialogTemplate == NULL)
    LPCDLGTEMPLATE m_lpDialogTemplate; // indirect if (m_lpszTemplateName
                                         // == NULL)
    void* m_lpDialogInit;      // DLGINIT resource data
    CWnd* m_pParentWnd;       // parent/owner window
    HWND m_hWndTop;           // top level parent window (may be disabled)

    _AFX_OCC_DIALOG_INFO* m_pOccDialogInfo;
    virtual BOOL SetOccDialogInfo(_AFX_OCC_DIALOG_INFO* pOccDialogInfo);
    virtual void PreInitDialog();
// implementation helpers
    HWND PreModal();
    void PostModal();

protected: // ** some omitted
    afx_msg LRESULT OnCommandHelp(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT HandleInitDialog(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT HandleSetFont(WPARAM wParam, LPARAM lParam);
    DECLARE_MESSAGE_MAP()
};

};
```

Here's a synopsis of what the undocumented data functions and members do. We will be exploring some of them in more detail later as we cover the various features of CDialog.

CDialog's Undocumented Data Members

The CDialog data members are used to store information passed to the constructor as well as CDialog initialization information. These members are protected, so that you can access them only through derived classes.

- `m_nIDHelp`—The help ID for the button, usually determined from the template ID.
- `m_lpszTemplateName`—The name of the resource template. If an ID was specified, `m_lpszTemplateName` will be `MAKELONG(0, ID)`.
- `m_hDialogTemplate`—The handle of the resource template once loaded.
- `m_lpDialogInit`—A pointer to the data that initializes the dialog. This data is loaded from the resource file and is associated with the dialog through its resource ID. We'll cover this member in much more detail because it controls the dialog initialization.
- `m_pParentWnd`—A CWnd pointer to the parent/owner window.
- `m_hWndTop`—The top-level parent window.
- `m_pOccDialogInfo`—Stores information needed for OLE controls.

CDialog's Member Functions of Interest

- `virtual PreTranslateMessage()`—Filters messages for special cases like tool tips and SHIFT-F1 context-sensitive help.
- `virtual OnCmdMsg()`—Routes command messages to the owner and current CWinThread object. Ignores control notifications.
- `virtual CheckAutoCenter()`—Checks to see if the user has specified that the dialog should be auto-centered by specifying one of the styles: DS_CENTER, DS_CENTERMOUSE, or DS_ABSALIGN. (Note: these are Windows 95 extension styles for dialogs.) Also, x and y must be zero.
- `virtual SetOccDialogInfo()`—Lets you set the `m_pOccDialogInfo` member data variable.
- `virtual PreInitDialog()`—A placeholder for CDialog derivatives. `PreInitDialog()` is called before WM_INITDIALOG (`OnInitDialog()`) and is useful if you want to perform very early initializations.
- `PreModal()`—Readies the CDialog for the `DoModal()` logic by preparing to attach itself to the created dialog window.
- `PostModal()`—Unhooks and detaches the CDialog after the `DoModal()` logic.

You will see many of these undocumented member functions pop up again as we travel through the key CDialog member functions.

Modal CDialog Creation

First, let's follow the life and times of a modal dialog through the MFC to get a feel for when the dialog is actually created and what implementation details are involved.

As noted in the CDialog review, usually you create a local CDialog derivative object and then just call DoModal(), so the two CDialog member functions of interest here are the constructor and the DoModal() call.

CDialog Construction

CDialog has two modal constructor overloads: one that takes a string template name as the first argument and another that takes the resource ID. The resource ID is used more often, so let's look at its implementation. Listing 6-2 contains the pseudocode for the modal CDialog constructor, from DLGCORE.CPP.

Listing 6-2. The CDialog constructor, from DLGCORE.CPP

```
CDialog::CDialog(UINT nIDTemplate, CWnd* pParentWnd)
{
    AFX_ZERO_INIT_OBJECT(CWnd);
    m_pParentWnd = pParentWnd; m_lpszTemplateName =
        MAKEINTRESOURCE(nIDTemplate);
    m_nIDHelp = nIDTemplate;
}
```

As you can see from Listing 6-2, the modal CDialog constructor zeroes out the instance (using the AFX_ZERO_INIT_OBJECT macro, as discussed in the sidebar) and then stuffs away the arguments into the corresponding CDialog data members.

The AFX_ZERO_INIT_OBJECT Macro

This handy MFC macro, which lives in AFX.H, can be used to zero-initialize your own classes that have data members. It is declared as follows:

```
#define AFX_ZERO_INIT_OBJECT(base_class) \
    memset(((base_class*)this)+1, 0, sizeof(*this) - \
    sizeof(class base_class));
```

The "+1" makes sure that the vtable does not get clobbered.

Note that using this macro is not a good idea if your class has any embedded members with construction semantics, such as embedded CString variables, other objects that have a specialized constructor, or objects that have virtual functions (since their vtbl ptrs will get clobbered).

DoModal()

Once a CDialog derivative is constructed, you should call DoModal(), which handles the dialog creation, display, and destruction for you. You might guess that this function is pretty big, based on the amount of work that it performs—and you'd be right! Listing 6-3 contains the pseudocode version of CDlg::DoModal(), from DLGCORE.CPP.

Listing 6-3. CDlg::DoModal() pseudocode, from DLGCORE.CPP

```
int CDlg::DoModal()
{
    // Code Block 1
    LPCDLGTEMPLATE lpDialogTemplate = m_lpDialogTemplate;
    HGLOBAL hDialogTemplate = m_hDialogTemplate;
    if (m_lpszTemplateName != NULL){
        HINSTANCE hInst = AfxFindResourceHandle(m_lpszTemplateName,
            RT_DIALOG);
        HRSRC hResource = ::FindResource(hInst, m_lpszTemplateName,
            RT_DIALOG);
        hDialogTemplate = LoadResource(hInst, hResource);
    }
    if (hDialogTemplate != NULL)
        lpDialogTemplate = (LPCDLGTEMPLATE)LockResource(hDialogTemplate);
    // return -1 in case of failure to load the dialog template resource
    if (lpDialogTemplate == NULL)
        return -1;
    // Code Block 2
    HWND hWndParent = PreModal();
    AfxUnhookWindowCreate();
    CWnd* pParentWnd = CWnd::FromHandle(hWndParent);
    BOOL bEnableParent = FALSE;
    if (hWndParent != NULL && ::IsWindowEnabled(hWndParent))
        ::EnableWindow(hWndParent, FALSE);
    bEnableParent = TRUE;
    // Code Block 3
    AfxHookWindowCreate(this);
    if (CreateDlgIndirect(lpDialogTemplate, CWnd::FromHandle(hWndParent)))
    {
        if (m_nFlags & WF_CONTINUEMODAL){
            // enter modal loop
            DWORD dwFlags = MLF_SHOWONIDLE;
            if (GetStyle() & DS_NOIDLEMSG) dwFlags |= MLF_NOIDLEMSG;
            VERIFY(RunModalLoop(dwFlags) == m_nModalResult);
        }
        SetWindowPos(NULL, 0, 0, 0, 0,
            SWP_HIDEWINDOW|SWP_NOSIZE|SWP_NOMOVE|SWP_NOACTIVATE|SWP_NOZORDER);
    }

    if (bEnableParent)
        ::EnableWindow(hWndParent, TRUE);
}
```

```

if (hWndParent != NULL && ::GetActiveWindow() == m_hWnd)
    ::SetActiveWindow(hWndParent);
// Code Block 4
DestroyWindow();
PostModal();
if (m_lpszTemplateName != NULL || m_hDialogTemplate != NULL)
    UnlockResource(hDialogTemplate);
if (m_lpszTemplateName != NULL)
    FreeResource(hDialogTemplate);
return m_nModalResult;
}

```

Because CDlg::DoModal() is such a long and in-depth member function, we've added comments (for example, // Code Block 1) that break it into easy-to-digest subsections.

- **Code Block 1:** Loading the dialog resource.

The first section of DoModal() uses the dialog template name to find, load, and lock the dialog template from the application's resource file. If DoModal() cannot locate the resource, it will return an error code of -1.

- **Code Block 2:** Preparing to create the dialog.

The second DoModal() section first calls PreModal(). PreModal() performs some safety checks (for example, makes sure that DoModal() wasn't called on a modeless dialog) and then finds the desired parent handle for the dialog. It determines this by calling CWnd::GetSafeOwner() and then stores the result in the m_hWndTop data member.

After calling PreModal(), the second section calls EnableWindow(FALSE) to disable the dialog's parent and stop it from getting any messages. This "freezes" the parent (usually the main window) and enforces modality.

- **Code Block 3:** Creating and displaying the dialog.

In section 3 of DoModal(), first DoModal() prepares to attach an MFC object to the dialog by calling AfxHookWindowCreate(). Then, DoModal() calls CWnd::CreateDlgIndirect(). This undocumented CWnd member function is located in DLGCORE.CPP. CWnd::CreateDlgIndirect() does some error checking and then sets up the fonts for the dialog. After setting the default fonts, CreateDlgIndirect() ORs in WF_CONTINUEMODAL into the CWnd::m_nFlags member and calls the Win32 API CreateDialogIndirect() with the dialog template, parent handle, and AfxDlgProc() as the dialog procedure. If all of the creation goes OK, CWnd::CreateDlgIndirect() returns TRUE; it returns FALSE if the dialog creation has failed.

If the CWnd::CreateDlgIndirect() call is successful, DoModal() then calls CWnd::RunModalLoop(). CWnd::RunModalLoop() processes messages until the

user presses either the OK or Cancel button. The OnOk() and OnCancel() member functions both call EndDialog(), which calls CWnd::EndModalLoop(), thus ending the CWnd::RunModalLoop() processing and returning the flow of control back to DoModal().

Once RunModalLoop() returns, DoModal() uses SetWindowPos() to hide the now-defunct dialog. After the dialog is hidden, DoModal() re-enables the parent window by calling EnableWindow(TRUE) and makes sure that the parent becomes the new active window by calling SetActivateWindow().

Understanding the third CDlg::DoModal() code block is very important to fully grasp CDlg. Be sure to check the DoModal() source and make sure you see what's going on in code block 3 before moving on. It may help you to fire up VC++ and set a breakpoint at CDlg::DoModal() for an about box in a sample and follow it that way.

- **Code block 4:** That's all, folks.

By the time we hit block 4, the modal dialog has been displayed, the user has hit OK or Cancel, and the dialog has been hidden. Now DoModal() calls DestroyWindow() to nuke the Windows dialog object and then calls PostModal(). PostModal() unhooks and detaches the C++ object from the Windows object and sets m_hWndTop to NULL.

Finally, DoModal() unlocks and frees any resources that need to be released and then returns the result that was returned by CWnd::RunModalLoop().

Phew! That's a lot of work for one member function. To summarize, the key to the DoModal() routine is the fact that it calls the Win32 API CreateDialogIndirect() and then uses CWnd::RunModalLoop() to process all of the messages and enforce the modality. Since the modal loop logic lives in CWnd, you can write your own CWnd derivatives and make them modal without having to derive from CDlg, if you so desire. The MFC property sheets that we will cover later are a perfect example of this.

Modeless CDlg Creation

Now that we've seen how a modal dialog is created, let's take the next logical step and follow modeless dialog creation. Remember that you usually create a modeless dialog using the new operator and then call CDlg::Create() to initialize and display the dialog. You can terminate the dialog by calling CDlg::EndDialog().

CDlg::Create()

The CDlg default constructor zero-initializes the class, so let's skip right to the action: the CDlg::Create() member function, from DLGCORE.CPP, as shown in Listing 6-4.

Listing 6-4. The CDlg::Create() implementation, from DLGCORE.CPP

```

BOOL CDlg::Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd)
{
    m_lpszTemplateName = lpszTemplateName;
    // used for help
    if (HIWORD(m_lpszTemplateName) == 0 && m_nIDHelp == 0)
        m_nIDHelp = LOWORD((DWORD)m_lpszTemplateName);

#ifdef _DEBUG
    if (!_AfxCheckDialogTemplate(lpszTemplateName, FALSE))
        ASSERT(FALSE);           // invalid dialog template name
    PostNcDestroy();          // cleanup if Create fails too soon
    return FALSE;
}

#endif // _DEBUG
HINSTANCE hInst = AfxFindResourceHandle(lpszTemplateName, RT_DIALOG);
HRSRC hResource = ::FindResource(hInst, lpszTemplateName, RT_DIALOG);
HGLOBAL hTemplate = LoadResource(hInst, hResource);
BOOL bResult = CreateIndirect(hTemplate, pParentWnd);
FreeResource(hTemplate);
return bResult;
}

```

First, Create() stores away the template name and the help IDs in their corresponding data members. Next, Create() checks that the template is valid for the case of dialogs, and if everything checks out, it finds and loads the resource.

Once the resource makes it through this gauntlet of checks, Create() calls CDlg::CreateIndirect() to create the dialog.

CreateIndirect() sets the parent window to the result of an AfxGetMainWnd() call and then calls our old friend CWnd::CreateDlgIndirect(), which you'll recall eventually calls the Win32 API CreateDialogIndirect(). Finally, Create() frees the loaded resource by calling FreeResource() and returns the result of the CreateIndirect() call.

At this point a modeless dialog is displayed, and the end user can start interacting with it. The modeless dialog will remain displayed until the user clicks either the OK or Cancel button, which eventually causes EndDialog() to be called. Or the application can call EndDialog() directly to terminate the modeless dialog.

An Interesting Technique from Create()

In CDlg::Create(), notice how there is some special checking in the DEBUG build. Create() calls the diagnostic helper _AfxCheckDialogTemplate(), which goes through and makes sure that the end user has supplied a valid dialog template by performing a variety of checks. This is a useful technique you can generalize and apply to your own classes.

If you have some kind of data that needs extra checking, instead of having it inline in multiple locations, move it to a diagnostic helper. Then, as you find new tests for the data, you only have to add them to the helper and not make multiple inline checks.

CDialog::EndDialog(): The Terminator

We briefly mentioned EndDialog() in the modal dialog discussion, but let's take a closer look. Listing 6-5 contains the source code for CDIALOG::EndDialog(), from DLGCORE.CPP.

Listing 6-5. CDIALOG::EndDialog: The CDIALOG terminator

```
void CDIALOG::EndDialog(int nResult)
{
    if (m_nFlags & (WF_MODALLOOP|WF_CONTINUEMODAL))
        EndModalLoop(nResult);
    ::EndDialog(m_hWnd, nResult);
}
```

In the modeless case, the WF_MODALLOOP flag will not be set, so EndModalLoop() will not be called. The Win32 API EndDialog() does the dirty deed of making sure the dialog is destroyed.

CDIALOG Control Initialization

MFC CDIALOG control initialization is so transparent that you may have never thought about investigating how it is implemented. Let's say you whip up a dialog using the VC++ resource editor and in the dialog is a combo box that you initialize with the strings "one", "two", and "three".

How does MFC get this information? A quick check of the generated code doesn't reveal any calls that look like this:

```
m_pCombo->AddString("one");
m_pCombo->AddString("two");
m_pCombo->AddString("three");
```

Perhaps if we trace the CDIALOG initialization we can discover how MFC performs its CDIALOG initialization magic act.

WM_INITDIALOG: Gotta Love It

When a dialog is created, Windows sends a WM_INITDIALOG message just before the dialog is displayed so that the application can initialize any controls in the dialog before they become visible to the end user.

The CDIALOG message map maps WM_INITDIALOG to CDIALOG::HandleInitDialog(). HandleInitDialog() performs some OLE control initializations and then causes the CDIALOG::OnInitDialog() routine to be called, which is the usual name for a WM_INITDIALOG handler.

CDialog::OnInitDialog() calls CWnd::ExecuteDlgInit(), which actually does the dialog initialization we are interested in.

Why CWnd::ExecuteDlgInit()?

At this point, you might be asking, "Why do so many of the CDialog-specific routines live in CWnd?" Good question! The reason is that there are other CWnd derivatives that do not derive from CDialog that need to use these routines, so instead of duplicating the code, the MFC team pushed them up into the common parent class, CWnd. While this does seem to be less object-oriented, it solves the problem of having tons of duplicate routines for handling modality and initialization. This will all make more sense after we cover MFC property sheets.

CWnd::ExecuteDlgInit()

There are actually two CWnd::ExecuteDlgInit() overloads. The first takes the dialog template name as the argument.

"Why would it do this?" you ask. It seems that ExecuteDlgInit() is going to do something with the dialog template, but that's not the case. If you look in an .RC file generated by VC++, you will notice a resource that looks like Listing 6-6:

Listing 6-6. DLGINIT resource data

```
IDD_ABOUTBOX DLGINIT
BEGIN
IDC_COMBO1, 0x403, 4, 0
0x6e6f, 0x0065,
IDC_COMBO1, 0x403, 4, 0
0x7774, 0x006f,
IDC_COMBO1, 0x403, 6, 0
0x6874, 0x6572, 0x0065,
IDC_COMBO1, 0x403, 5, 0
0x6f66, 0x7275, "\000"
IDC_COMBO1, 0x403, 5, 0
0x6966, 0x6576, "\000"
IDC_COMBO1, 0x403, 4, 0
0x6973, 0x0078,
IDC_COMBO1, 0x403, 6, 0
0x6573, 0x6576, 0x006e,
0
END
```

This user-defined resource lets you put raw data into a resource file that you give your own name. In this example, the name is DLGINIT and the resource ID is the same as the dialog that the DLGINIT data belongs to, IDD_ABOUTBOX. MFC knows to expect this data, and the first ExecuteDlgInit() finds the DLGINIT resource for the

named dialog. If a DLGINIT resource exists for the named dialog, ExecuteDlgInit() loads and locks the resource to get a pointer to the raw data. Once it has the pointer, it calls the second ExecuteDlgInit() override, which parses the raw data.

ExecuteDlgInit(): Parsing the DLGINIT Data

Now that we know “where” MFC puts the initialization information, the big mystery is its format. Let’s look at ExecuteDlgInit() and see what it does with the raw DLGINIT data. Listing 6-7 shows an abbreviated version of ExecuteDlgInit() from WINCORE.CPP. Note that any code that initializes OLE controls has been omitted for brevity.

Listing 6-7. CWnd::ExecuteDlgInit(), from WINCORE.CPP

```
BOOL CWnd::ExecuteDlgInit(LPVOID lpResource)
{
    BOOL bSuccess = TRUE;
    UNALIGNED WORD* lpnRes = (WORD*)lpResource;
    while (bSuccess && *lpnRes != 0){
        WORD nIDC = *lpnRes++;
        WORD nMsg = *lpnRes++;
        DWORD dwLen = *((UNALIGNED DWORD*)&lpnRes)++;
        if (nMsg == LB_ADDSTRING || nMsg == CB_ADDSTRING){
            if (::SendDlgItemMessageA(m_hWnd, nIDC, nMsg, 0,
                (LONG)lpnRes) == -1)
                bSuccess = FALSE;
        }
        // skip past data lpnRes = (WORD*)((LPBYTE)lpnRes + (UINT)dwLen);
    }
    return bSuccess;
}
```

ExecuteDlgInit() uses a WORD pointer, lpnRes, to iterate through the raw data. First, it extracts an ID into nIDC. Next, it extracts a message into nMsg. Finally, it extracts a DWORD into dwLen.

If the extracted nMsg is an LB_ADDSTRING or CB_ADDSTRING, ExecuteDlgInit() calls SendDlgItemMessage() with the extracted data.

Putting the Pieces Together

Now we have all the pieces to the MFC dialog control initialization puzzle and can put them together to see the solution. Visual C++ stores the initialization in the resource file as raw data under the name DLGINIT and with the same resource ID as the dialog it belongs to.

At dialog creation in response to WM_INITDIALOG, MFC loads this resource, which contains control messages. ExecuteDlgInit() iterates through the raw data,

decodes the messages, and sends them to the already created but not yet visible dialog controls.

Pretty neat, eh!? A side benefit from this approach is that the strings do not take up any memory in the data part of the executable, and they can be localized more easily, since they live in the resource file.

Keep in mind that OLE controls are also initialized using the same technique. We'll talk more about OLE control internals in Chapter 15.

MFC supports another way to initialize dialog control that also lets you easily exchange information between member variables and controls, as well as validate end-user dialog control input. This MFC feature is called dynamic data exchange and dialog data validation, or DDX/DDV.

DDX/DDV: CDlg Exchange and Validation

To see how DDX/DDV is implemented, we'll first review how to use DDX/DDV in your applications, and then we'll look at some MFC code snippets that explain how this handy feature works under the hood.

Using DDX/DDV

As the name implies, DDX/DDV is actually two features. First, there's dynamic data exchange. DDX "wires" a data member to a control so that when a dialog is displayed, you can specify the initial value in the data member, and it is copied to the control when the dialog is shown. Also, when the dialog is destroyed, the state of the control is copied back into your member variable. For example, if you have a two-state checkbox, it is advantageous to "wire" this control to a Boolean member variable so that you do not have to constantly call GetCheck() and save the value.

To use DDX in your dialogs, you declare a DoDataExchange() member function that takes a CDataExchange pointer. Next, you add calls to control-specific functions that map the controls to your data members. Listing 6-8 shows an example of a dialog that has two edit controls, a checkbox, and a radio button.

Listing 6-8. An example CDlg-derivative DoDataExchange() member function

```
void CMYDlg::DoDataExchange(CDataExchange * pDX)
{
    CDlg::DoDataExchange(pDX); //Call base first.
    DDX_Text(pDX, IDC_EDIT1, m_strEdit1);
    DDX_Text(pDX, IDC_EDIT2, m_uEdit2);
    DDX_Check(pDX, IDC_CHECK, m_bCheckState);
    DDX_Radio(pDX, IDC_RADIO, m_nRadioState);
}
```

In the example, the edit with resource ID IDC_EDIT1 is wired to a CString member variable, m_strEdit1. The edit with ID of IDC_EDIT2 is wired with the unsigned integer member variable m_uEdit2. Similarly, the checkbox control with resource ID IDC_CHECK is wired to the Boolean member variable, m_bCheckState, and the radio button with resource ID IDC_RADIO is wired to the m_nRadioState member variable.

There's at least one DDX function for each Windows control type, plus there are multiple overloads per control type, so that you can use different member variable types with the controls. The second DDX_Text call in Listing 6-8 is an example of an overloaded DDX function that automatically converts the string in an edit into an integer for you.

If it seems as if writing the DoDataExchange() routine could get tedious, you're right! Luckily the Visual C++ Class Wizard takes care of both adding the DoDataExchange() member function and calling the functions that map the controls to your data members. Just check out the "Member Variable" tab in Class Wizard and you will see how to add entries to the DoDataExchange() member function. Figure 6-2 shows the Class Wizard Member Variable tab.

Once you have implemented your DoDataExchange() member function, MFC takes care of the rest. If you want to initialize some controls, you just initialize the corresponding data members in your CDialog derivative's constructor and the controls will have the desired state when displayed. When your dialog is dismissed via the OK button, the member variables will magically contain the state of the dialog controls.

The second MFC dialog feature is dialog data validation. This feature works in tandem with DDX to validate the data being exchanged. For example, if you want the user to enter a number between 1 and 234 in an edit box, DDV makes sure that what the user enters fits the restrictions of being an integer and falling between 1 and 234.

To use DDV support, you use one of the DDV functions that specify the restrictions you are placing on a control/data member pair.

Let's extend the previous example to force the user to enter a certain number of characters in the first edit control and a number between 1 and 234 in the second edit control. Listing 6-9 shows how we would do this.

Listing 6-9. Adding DDV control validation to the DDX example

```
void CMyDlg::DoDataExchange(CDataExchange * pDX)
{
    CDialog::DoDataExchange(pDX); //Call base first.
    DDX_Text(pDX, IDC_EDIT1, m_strEdit1);
    DDV_MaxChars(pDX, m_strEdit1, 20);
    DDX_Text(pDX, IDC_EDIT2, m_nEdit2Integer);
    DDV_MinMaxUInt(pDX, m_uEdit2, 1, 234);
    DDX_Check(pDX, IDC_CHECK, m_bCheckState);
    DDX_Radio(pDX, IDC_RADIO, m_nRadioState);
}
```

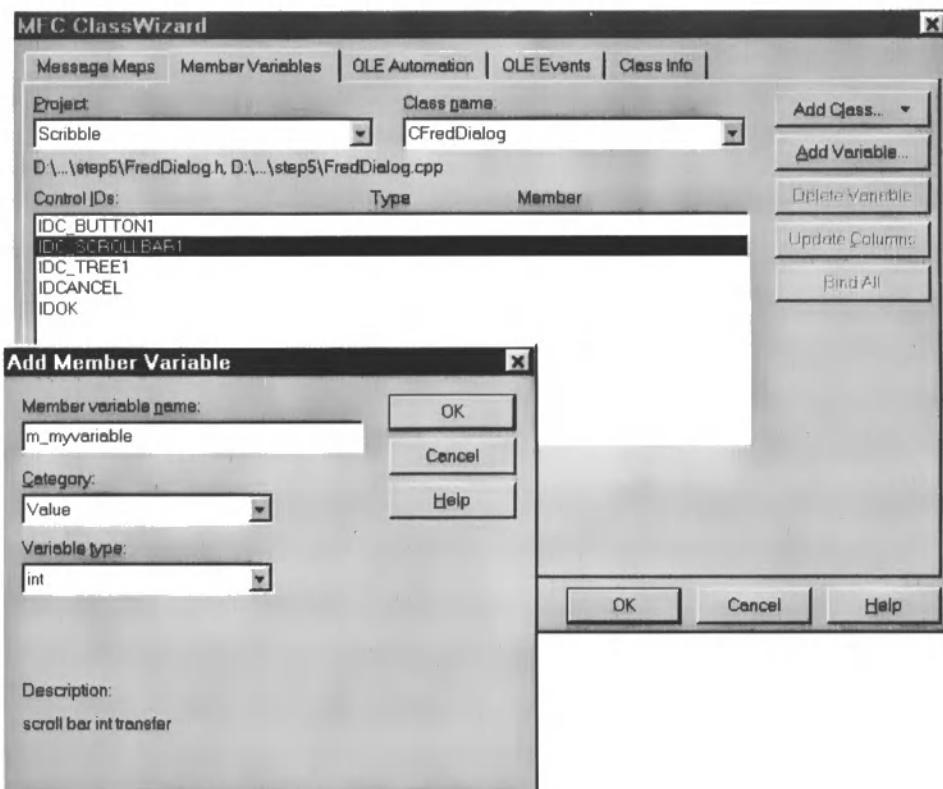


Figure 6-2. The Class Wizard Member Variable tab

That's all you have to do to enforce limits on the user. Now let's see how MFC accomplishes all of the DDX/DDV functionality.

How DDX/DDV Works

This review of how to use DDX/DDV raises some big questions about how it works:

- What is the role of CDataExchange in all of this?
- When, where, and how does MFC get the control information to and from the member variables?
- How do bumblebees fly?
- What are the DDX/DDV functions doing?

Let's take these questions one at a time (leaving the third as a reader exercise) and start by dissecting that new helper class, **CDataExchange**.

The DDX/DDV Helper Class, CDataExchange

The declaration for the DDX/DDV helper class lives in AFXWIN.H and is reproduced in Listing 6-10 for your reference.

Listing 6-10. The declaration for the DDX/DDV helper class, CDataExchange

```
class CDataExchange
{
// Attributes
public:
    BOOL m_bSaveAndValidate;    // TRUE => save and validate data
    CWnd* m_pDlgWnd;          // container usually a dialog
// Operations (for implementors of DDX and DDV procs)
    HWND PrepareCtrl(int nIDC);      // return HWND of control
    HWND PrepareEditCtrl(int nIDC);   // return HWND of control
    void Fail();                  // will throw exception
    CWnd* PrepareOleCtrl(int nIDC); // for OLE controls in dialog
// Implementation
    CDataExchange(CWnd* pDlgWnd, BOOL bSaveAndValidate);
    HWND m_hWndLastControl;        // last control used (for validation)
    BOOL m_bEditLastControl;       // last control was an edit item
};
```

Here's a brief introduction to what each member in **CDataExchange** does. If you are curious about the details of these members, you can find them in the DLGDATA.CPP MFC source file.

CDataExchange Member Functions

- Constructor—Passes in the dialog pointer and the initial **bSaveAndValidate** setting.
- **PrepareCtrl()**—Prepares a non-edit control for DDX/DDV. Basically this function gets the window handle to the control by calling **GetDlgItem()** with the resource ID of the control on the internal **m_pDlgWnd** dialog pointer.
- **PrepareEditCtrl()**—Prepares an edit control for DDX/DDV. First calls **PrepareCtrl()** and then sets **m_bEditLastControl** to **TRUE**, where **PrepareCtrl()** sets it to **FALSE**.
- **Fail()**—Called when validation fails. **Fail()** resets the focus back to the previous control and throws a **CUserException** exception.
- **PrepareOleCtrl()**—Prepares an OLE control for DDX/DDV.

CDataExchange Member Variables

- **m_bSaveAndValidate**—This member specifies the direction of the DDX and if it should do DDV. If FALSE, data is flowing from member variables to controls. If TRUE, data is flowing from the controls to the member variables. DDV only happens when this flag is TRUE.
- **m_pDlgWnd**—A CWnd pointer to the dialog that contains the controls to be exchanged and validated. This value is passed in through the CDataExchange constructor.
- **m_hWndLastControl**—m_hWndPreviousControl would be a better name for this CDataExchange member variable, which keeps track of the handle of the previous control.
- **m_bEditLastControl**—A Boolean that specifies if the previous control was an edit.

As we continue our sleuth work on the DDX/DDV internals, the roles of these CDataExchange members will become much clearer.

Inside the DDX/DDV Routines

Now that you're familiar with the members of CDataExchange, we can look at some DDX/DDV routines and see firsthand what they are doing.

Let's look at the routines used by the example in Listing 6-9. First we called DDX_Text with a CString to exchange data between a CString and an edit control. Listing 6-11 shows the MFC pseudocode for that function, from DLGDATA.CPP, the home of the DDX/DDV routines.

Listing 6-11. The DDX_Text() routine pseudocode, from DLGDATA.CPP

```
void DDX_Text(CDataExchange* pDX, int nIDC, CString& value)
{
    HWND hWndCtrl = pDX->PrepareEditCtrl(nIDC);
    if (pDX->m_bSaveAndValidate){
        int nLen = ::GetWindowTextLength(hWndCtrl);
        ::GetWindowText(hWndCtrl, value.GetBufferSetLength(nLen), nLen+1);
        value.ReleaseBuffer();
    }
    else
        AfxSetWindowText(hWndCtrl, value);
}
```

Every DDX routine first calls either PrepareEditCtrl() or PrepareCtrl(), depending on the control type. Next, DDX_Text() checks the m_bSaveAndValidate Boolean value. If this flag is TRUE (remember that that means that data is flowing from the control to the member variable), then DDX_Text() calls the Win32 APIs for getting

the string from the edit control and places them in the “value” argument, which in our example would be the `m_strEdit1` member variable.

If `CDataExchange::m_bSaveAndValidate` is FALSE (data is flowing from the member variable to the control), then `DDX_Text()` updates the edit control with the contents of the `CString` argument using an AFX convenience routine. Figure 6-3 graphically illustrates the flow of DDX/DDV data.

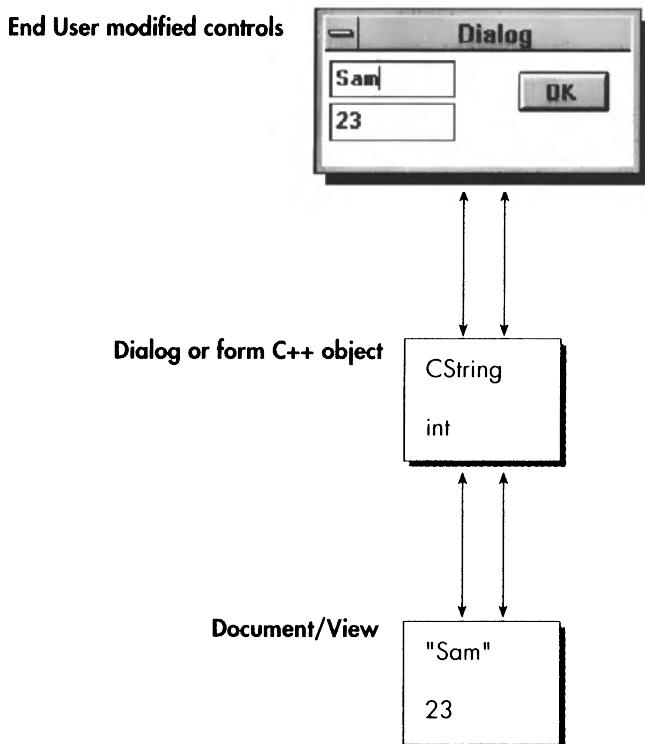


Figure 6-3. DDX/DDV flow of control

AfxSetWindowText()

Throughout MFC, you will notice that they use the `AfxSetWindowText()` utility function instead of directly calling `SetWindowText()`. Why? Well, looking at the implementation of `AfxSetWindowText()` in `WINUTIL.CPP`, you will notice that it first checks, using `GetWindowText()`, if the text is actually different before setting the text. It does this to reduce time-consuming updates and also to keep the controls from flashing. You should consider using this utility function yourself for the same reasons that the MFC team uses it.

Next, let's look at the DDV_MaxChars() validation routine from DLGDATA.CPP, in Listing 6-12, which verifies the number of characters entered do not exceed the specified maximum.

Listing 6-12. The DDV_MaxChars() pseudocode, from DLGDATA.CPP

```
void DDV_MaxChars(CDataExchange* pDX, CString const& value, int nChars)
{
    if (pDX->m_bSaveAndValidate && value.GetLength() > nChars){
        TCHAR szT[32]; wsprintf(szT, "%d", nChars);
        CString prompt;
        AfxFormatString1(prompt, AFX_IDP_PARSE_STRING_SIZE, szT);
        AfxMessageBox(prompt, MB_ICONEXCLAMATION, AFX_IDP_PARSE_STRING_SIZE);
        prompt.Empty(); // exception prep
        pDX->Fail();
    }
    else if (pDX->m_hWndLastControl != NULL && pDX->m_bEditLastControl)
        ::SendMessage(pDX->m_hWndLastControl, EM_LIMITTEXT, nChars, 0);
}
```

First, DDV_MaxChars() checks to make sure that the CDataExchange::m_bSaveAndValidate flag is TRUE, indicating that it is time for validation. DDV_MaxChars() then checks the length of the string against the specified ceiling. If the length of the string is valid, then nothing will happen. If the length of the string is above the ceiling, then the block of code under the “if” statement is executed.

This block of code prompts the user via a message box (by calling AfxMessageBox()) that the string is too large. The actual message displayed is loaded from a resource by the AfxFormatString1 utility function. In this example, the string, whose ID is AFX_IDP_PARSE_STRING_SIZE and which lives in AFXRES.RC, is “Please enter no more than %1 characters”. MFC places all end-user messages inside resources so that they can be localized easily. AfxFormatString1() automatically replaces the %1 with the specified size, szT, or 20, as in our example from Listing 6-9.

Once the message box has been displayed, DDV_MaxChars() calls CDataExchange::Fail(), which shifts the focus back to the control in question and throws a CUserException exception.

The second half of DDV_MaxChars() verifies that it has a non-NULL handle to an edit and then uses the EM_LIMITTEXT edit message to tell the edit to limit the size of the input. This shifts the burden of limiting the number of characters to the Windows control and not the DDV routines. Most of the other DDX/DDV routines take very similar approaches.

One important implementation helper is the DDX_TextWithFormat() function. This function is called to convert strings into integers and vice versa. The implementation can be found in DLGDATA.CPP and makes for interesting research. See if you can

figure out how the AfxSimpleScanf() utility function works and find any limitations it might have.

Unanswered Questions

There's certainly nothing magical about the DDX/DDV routines once you know the role of the CDataExchange class helper members. There are still some questions about DDX/DDV that are unanswered:

- When does CDialog::DoDataExchange() get called and by whom?
- Where does CDataExchange get created?
- What "triggers" DDX/DDV to happen?
- Who catches these exceptions that CDataExchange::Fail() is throwing?

The answers to all of these questions and more can be traced to one key member function.

Digging for the Answers

We know that there are a couple of places that DoDataExchange() will need to be called. Most likely these would be dialog initialization to initialize the controls with the member function information and before the dialog is dismissed.

Looking at the CDialog member functions OnInitDialog() and OnOK(), you will notice that they call CWnd::UpdateData() with FALSE and TRUE, respectively.

Listing 6-13 shows the pseudocode, from DLGCORE.CPP, for the default CDialog::OnOK() member function.

Listing 6-13. Pseudocode for CDialog::OnOK(), from dlgcore.cpp

```
void CDialog::OnOK()
{
    if (!UpdateData(TRUE))
        TRACE0("UpdateData failed during dialog termination.\n");
    EndDialog(IDOK);
}
```

Aha! This looks as if it could be the clue that answers the remaining DDX/DDV implementation questions. OnOK() is calling UpdateData(TRUE) and if the result is not TRUE, it does not close out the dialog. Let's see what's going on inside UpdateData() and how it fits into the DDX/DDV puzzle.

CWnd::UpdateData(): The Missing Link Unearthed!

CWnd::UpdateData() is another one of the dialog-specific routines that have been pushed up into CWnd so that other MFC CWnd derivatives that contain controls can support DDX/DDV. (Other examples are CWnd::RunModalLoop() and CWnd::ExecuteDlgInit().)

Let's take a look at the pseudocode for CWnd::UpdateData() from WINCORE.CPP to see what this pivotal routine is up to. Listing 6-14 contains the pseudocode.

Listing 6-14. The CWnd::UpdateData() pseudocode, from WINCORE.CPP

```
BOOL CWnd::UpdateData(BOOL bSaveAndValidate)
{
    // Code Block 1
    CDataExchange dx(this, bSaveAndValidate);
    // Code Block 2
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    HWND hWndOldLockout = pThreadState->m_hLockoutNotifyWindow;
    pThreadState->m_hLockoutNotifyWindow = m_hWnd;
    // Code Block 3
    BOOL bOK = FALSE;           // assume failure
    TRY {
        DoDataExchange(&dx);
        bOK = TRUE;           // it worked
    }
    CATCH(CUserException, e) {
        // Code Block 4
        ASSERT(bOK == FALSE);
    }
    AND_CATCH_ALL(e) {
        //Code Block 5
        e->ReportError(MB_ICONEXCLAMATION, AFX_IDP_INTERNAL_FAILURE);
        ASSERT(!bOK);
        DELETE_EXCEPTION(e);
    }
    END_CATCH_ALL
    //Code Block 6
    pThreadState->m_hLockoutNotifyWindow = hWndOldLockout;
    return bOK;
}
```

This is a pretty lengthy routine, so to help explain it, we've broken it into code blocks, as described next.

- Code Block 1: First, UpdateData() creates a CDataExchange instance (so that's where it gets created!) and passes in this as the CDialo pointer and passes through the bSaveAndValidate flag. (Remember that OnDialogInit() calls this with FALSE, meaning that data flows from the member variables to controls; OnOK() specifies TRUE and causes the data to flow from control to member variable *and* validation to occur.)
- Code Block 2: Next, UpdateData() blocks the current thread from receiving control notifications. (Chapter 10 goes into threads in more detail.)

- Code Block 3: The local Boolean bOK is used to store the results of the DDX/DDV code. Once UpdateData() sets bOK to FALSE (a very pessimistic approach), it calls your CDlg derivative's DoDataExchange with the CDataExchange instance as argument. This call is made inside of a TRY block in case CDataExchange::Fail() is called within a DDV function and throws an exception. If an exception is not tripped, bOK is set to TRUE, which is returned from OnUpdate(), which signals that all fields have been exchanged and validated.
- Code Block 4: This block is called in when a CUserException is tripped, which is the case when CDataExchange::Fail() is called. At this point bOK should be FALSE, and the exception will just fall through and UpdateData() will return 0.

The CUserException exception is a special exception type which indicates that the user has already received a message about the error, meaning that no message box is necessary at the catch site.

Also in Code Block 4, you will notice the check ASSERT(bOK == FALSE).

But based on the logic of the function, this could never be the case. MFC uses this ASSERT as a sanity check, just to make sure the logic of the routine is flowing as expected. You might want to consider similar sanity checks in your own MFC code, especially if you have a team of developers working on the same body of code.

- Code Block 5: Block 5 is called if any other exception has tripped during the DDX/DDV process. It reports that a serious error has occurred and also causes FALSE to be returned.
- Code Block 6: The final block resets the notify window so that control notifications are processed in the usual way and finally returns either FALSE or TRUE, depending on the results of DoDataExchange().

For modal dialogs, it makes sense to call UpdateData() in OnInitDialog() and OnOK(), but with a modeless dialog, you might want DDX/DDV to occur when the user presses a button (for example, “Apply”) or in response to some other end-user interaction. Now you know that you can just call UpdateData(TRUE) to trigger DDX/DDV for your modeless dialog.

Putting It All Together

To tie together all of these DDX/DDV revelations, imagine the following scenario:

There's a modal dialog with an edit field that accepts integers between 1 and 234. The implementation of the dialog's DoDataExchange() member function contains a DDX_Text() and a DDV_MinMaxUInt() call. The DDX call wires the edit control to the m_uEditValue member.

This not-uncommon scenario results in the following chain of events:

During dialog creation, `OnInitDialog()` is called, which calls `UpdateData(FALSE)`. `UpdateData()` calls the dialog's `DoDataExchange()` specifying FALSE in the `CDataExchange::m_bSaveAndValidate` member. The `DDX_Text()` routine is called. `DDX_Text()` checks the `m_bSaveAndValidate` flag and detects that it should initialize the edit control with the contents of `m_uEditText`. It calls `AfxSetWindowText()` to accomplish this.

Next, the `DDV_MinMaxUInt()` routine is called. It checks `m_bSaveAndValidate` and determines that it should do nothing because the flag is FALSE.

At this point, all DDX/DDV is done, `DoDataExchange()` finishes, and the dialog is created.

Now, imagine that the user enters "356" into the edit. Once the user presses the OK button, `CDialog::OnOK()` is called, which calls `UpdateData(TRUE)`.

`UpdateData()` calls `DoDataExchange()`, which then calls our `DDX_Text()` routine, which sees that `m_bSaveAndValidate` is TRUE, so it converts the string "356" into an unsigned integer and copies the value into the `m_uEditText` member variable. Next, the `DDV_MinMaxUInt()` routine in `DoDataExchange()` is called. `DDV_MinMaxUInt()` detects that it should validate and finds that there's a problem. To resolve the problem, `DDV_MinMaxUInt()` displays a message box saying, "Please enter an integer between 1 and 234". After the user dismisses the message box, the `DDV_MinMaxUInt()` routine calls `CDataExchange::Fail()`.

`CDataExchange::Fail()` sets the focus to the edit control in question and then throws an exception. This exception notifies `UpdateData()` that there has been a problem and it returns FALSE to `OnOK()`. When `OnOK()` realizes that `UpdateData(TRUE)` has failed, it does not call `EndDialog()`, thus leaving the dialog up so that the end user can make corrections to the selected validation-failed control and try again.

Wow! Quite a bit of work going on in those seemingly simple DDX/DDV routines, eh?

Now we've pretty much exhausted the interesting points of the `CDialog` class. Let's take a look at some of `CDialog`'s derivatives, the MFC common dialog classes, and see how they use and enhance the now-familiar `CDialog` interface.

MFC Common Dialogs

Microsoft introduced the common dialogs in Windows 3.1 to provide a "standard" user interface for common operations such as opening/saving files, choosing fonts, find/replace, printing, and choosing colors. With the release of Windows 95, the common dialogs got a major facelift and now have a Windows 95 Explorer look and feel.

The MFC common dialogs wrap the Windows common dialog APIs and give them a CDialog interface (for example, DoModal() and OnInitDialog()). Table 6-1 lists the MFC common dialog classes and the operations they perform.

Table 6-1. The MFC common dialog classes

MFC Class Name	Operation
CColorDialog	Lets the user choose a color.
CFileDialog	Used for opening and saving files.
CFindReplaceDialog	Allows the user to find and replace items.
CFontDialog	Provides a font-choosing dialog.
CPrintDialog	Lets the user choose a printer, number of pages, orientation, and so on.
CPageSetupDialog	A Win95 common dialog that replaces CPrintDialog and adds some page setup options, such as page margins.

Versions Note

The MFC common dialog classes have been in MFC since version 1.0 (they were officially documented in 2.0). The exception is CPageSetupDialog, which was introduced in MFC 4.0.

Let's quickly review how to use the common dialogs and then see how they are implemented in MFC.

Using the MFC Common Dialog Classes

The steps for using the MFC common dialog classes are these:

1. Call the constructor, passing in arguments that tailor the common dialog to your needs. (For example, you can specify file filters and a variety of other flags.)
2. Call DoModal().
3. If the return value from DoModal() is IDOK, you can then call some of the class's member functions to retrieve the information that the user has specified.

Listing 6-15 is a brief example that shows how to create a File Open dialog.

Listing 6-15. An example usage of the MFC common dialog classes

```
CFileDialog myDialog(TRUE,NULL,NULL,OFN_FILEMUSTEXIST);
if (myDialog.DoModal() == IDOK) {
    //Now we can get the filename, path and others.
```

```

    CString strPath = myDialog.GetPathName();
    CString strFile = myDialog.GetFileName();
}
else
    //User selected cancel...

```

Though this example focuses on the CFileDialog, the other MFC common dialog classes work just the same way, but return colors, fonts, and other relevant information.

You can customize the Windows 3.x common dialogs by providing a new dialog template and also by specifying a hook (callback) that will intercept key messages. The Windows 95 Explorer-style common dialogs also have this capability, but instead of providing an entirely new dialog template, you just create a template that contains the controls you want to add to the dialog.

To super-customize using the MFC common dialog classes, you create a common dialog derivative class and in the constructor set flags in the OPENFILENAME structure. For example, you would set OFN_ENABLETEMPLATE to specify that you would like to use your own template. Next, you set a field of the OPENFILE structure to the handle of your dialog template.

Instead of providing a hook, you just have to override some virtual functions to intercept various events. For example, if you want to check the name of a file, you can override OnFileNameOK() and check any file name before the dialog is dismissed.

That's the end of the quick refresher on using the MFC common dialog classes. Now let's dive into the MFC source code and learn more about them.

Common Dialogs Internals

To understand how the MFC common dialog classes work, you need to be aware of how the Windows common dialogs are used from a Win32 API standpoint. Each dialog has a structure whose fields describe the settings for the dialog. Each dialog also has a corresponding API that takes the structure as argument, displays the dialog, interacts with the user, and returns. For example, with the Open File common dialog, the structure is OPENFILENAME and the API is GetOpenFileName(LPOPENFILENAME).

Each of these structures and functions is heavily documented in almost every Windows programming book, so we'll skip the gory details here. The main point is that the lion's share of the work is going on inside of the Win32 common dialog function. With this in mind, let's look at what part the MFC common dialog classes play.

The MFC common dialog classes are declared in the AFXDLGS.H header file, and their implementations are spread among a variety of source files. Table 6-2 shows you where to look for the implementation of each MFC common dialog class.

Table 6-2. Where to look for common dialog source

Class Name	Source File Name
CCo lorDialog	DLGCLR.CPP
CFileDialog	DLGFILE.CPP
CFindReplaceDialog	DLGFR.CPP
CFontDialog	DLGFNT.CPP
CPrintDialog	DLGPRNT.CPP
CPageSetupDialog	DLGPRNT.CPP

While looking in the AFXDLGS.H header file, you will notice that all of the common dialogs actually derive from CCommonDialog, instead of from CDialog. Let's take a look at this common dialog base class.

CCommonDialog: The Common Dialog Base Class

CCommonDialog is a very small class that defines just two virtuals, which every common dialog inherits. Listing 6-16 shows CCommonDialog's declaration, from AFXDLG.H.

Listing 6-16. CCommonDialog's declaration, from AFXDLGS.H

```
class CCommonDialog : public CDialog
{
public:
    CCommonDialog(CWnd* pParentWnd);
// Implementation
protected:
    virtual void OnOK();
    virtual void OnCancel();
};
```

CCommonDialog provides a holding place for the common OnOK() and OnCancel() handling for all of the common dialogs. Let's look at the implementation of the CCommonDialog::OnOK() function, from DLGCOMM.CPP, to see what it does (Listing 6-17).

Listing 6-17. CCommonDialog::OnOK()'s implementation

```
void CCommonDialog::OnOK()
{
    if (!UpdateData(TRUE)) {
        TRACE0("UpdateData failed during dialog termination.\n");
        // the UpdateData routine will set focus to correct item return;
}
```

```

// Common dialogs do not require ::EndDialog
Default();
}

```

`CCommonDialog::OnOK()` first calls our old friend `UpdateData()`. While none of the MFC `CCommonDialog` derivatives have `DoDataExchange()` member functions, this `UpdateData()` call makes sure that any of your classes that have `CCommonDialog` in the hierarchy have their `DoDataExchange()` members called. The call to `Default()` lets the common dialogs decide if they should call `EndDialog()` if necessary.

Exploring CFileDialog

The common dialogs all have very similar implementations, so instead of looking at all of them, we'll look at the `CFileDialog` common dialog. The details we uncover in `CFileDialog` can be applied to all of the other common dialog classes. An abbreviated declaration for `CFileDialog`, from `AFXDLGS.H`, is in Listing 6-18.

Listing 6-18. The `CFileDialog` class declaration, from `AFXDLGS.H`

```

class CFileDialog : public CCommonDialog
{
public:
// Attributes
OPENFILENAME m_ofn; // open file parameter block
// Constructors **omitted
// Operations **omitted
// Overridable callbacks
protected:
    friend UINT CALLBACK _AfxCommDlgProc(HWND, UINT, WPARAM, LPARAM);
    virtual UINT OnShareViolation(LPCTSTR lpszPathName);
    virtual BOOL OnFileNameOK();
    virtual void OnLBSelChangedNotify(UINT nIDBox, UINT iCurSel, UINT nCode);
    // only called back if OFN_EXPLORER is set
    virtual void OnInitDone();
    virtual void OnFileNameChange();
    virtual void OnFolderChange();
    virtual void OnTypeChange();

// Implementation
protected:
    BOOL m_bOpenFileDialog;      // TRUE for file open, FALSE for file save
    CString m_strFilter;         // filter string
    TCHAR m_szFileTitle[64];     // contains file title after return
    TCHAR m_szFileName[_MAX_PATH]; // contains full path name after return
    OPENFILENAME* m_pofnTemp;
    virtual BOOL OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult);
};

```

In the CFileDialog // Attributes subsection, you will notice that there is a public OPENFILENAME member, m_ofn. This is the structure that gets passed to GetOpenFileName(). Most of the CFileDialog code manipulates this structure in some form or another.

The next interesting part of the CFileDialog declaration is the set of protected virtual functions. If you compare this list to the documented list (at least in MFC 4.0), you will notice that there are quite a few undocumented callbacks in this class. We'll look at these in detail later. Also notice that the helper function, _AfxCommDlgProc(), is declared as a friend to the class.

Finally, in the // Implementation section of the class, we see that there are some fairly standard buffers that store things like the filter, file title, and file name. After DoModal(), when the user calls something like GetFileName(), these buffers will contain the requested information.

Also in the // Implementation section, there is a mysterious OPENFILENAME pointer, m_pofnTemp, and a virtual OnNotify() member function.

Before we explore some of these CFileDialog mysteries, let's look at the implementation of the constructor and DoModal() just so we can get a feel for how the common dialogs are created and displayed before moving on.

CFileDialog Construction and DoModal()

The CFileDialog constructor initializes the implementation members and then processes the constructor arguments and converts them into the appropriate OPENFILENAME settings for m_ofn.

In addition, if the constructor is called on Windows 95, it sets the OFN_EXPLORER flag and OFN_ENABLEHOOK flags. OFN_EXPLORER makes sure that the dialog is created with the Windows 95 look and feel. The OFN_ENABLEHOOK flag enables MFC to provide a hook routine to the common dialog. After setting the OFN_ENABLEHOOK flag, the CFileDialog constructor sets the hook field in m_ofn to _AfxCommDlgProc().

Because the m_ofn structure is public, you can change any of the settings made in the constructor in your derivative or directly to an instance of CFileDialog.

With the actions of the CFileDialog constructor in mind, let's look at the pseudocode implementation of CFileDialog::DoModal() as presented in Listing 6-19.

Listing 6-19. CFileDialog::DoModal() pseudocode, from DLGFILE.CPP

```
int CFileDialog::DoModal()
{
    int nResult;
    BOOL bEnableParent = FALSE;
    m_ofn.hwndOwner = PreModal();
    AfxUnhookWindowCreate();
```

```

if (m_ofn.hwndOwner != NULL && ::IsWindowEnabled(m_ofn.hwndOwner)){
    bEnableParent = TRUE;
    ::EnableWindow(m_ofn.hwndOwner, FALSE);
}
if (m_bOpenFileDialog)
    nResult = ::GetOpenFileName(&m_ofn);
else
    nResult = ::GetSaveFileName(&m_ofn);
if (bEnableParent)
    ::EnableWindow(m_ofn.hwndOwner, TRUE);
PostModal();
return nResult ? nResult : IDCANCEL;
}

```

First, CFileDialog::DoModal() calls CDialog::PreModal() and stores the handle to the owner window in the m_ofn.hWndOwner field.

Next, DoModal() disables the parent if there is one, to enforce modality.

After disabling the parent window, DoModal() checks the m_bOpenFileDialog member variable set by the CFileDialog constructor, and if it is TRUE, calls the Win32 API ::GetOpenFileName() because the user wants the Open File common dialog.

If m_bOpenFileDialog is FALSE, this indicates to DoModal() that the class user wants the Save File dialog, so it calls the Win32 API ::GetSaveFileName(). DoModal() passes in the m_ofn data member to both of these common dialog API calls.

At this point, the common dialog will be displayed until the user hits either the OK or Cancel button. The return value from GetOpenFileName()/GetSaveFileName() is stored in the local nResult variable.

Once the common dialog APIs have returned, DoModal() re-enables the parent window if it was disabled, calls PostModal(), and returns the result if it was greater than zero and IDCANCEL if it was not.

Where's the Meat?!

So far the CFileDialog has been kind of predictable. The constructor sets the fields of m_ofn, DoModal() calls the common dialog API, and that's about it. Don't give up hope, because there is some interesting undocumented behavior going on in the logic that provides those virtual callback routines we saw in the CFileDialog declaration.

The key to some of the callbacks is the hook procedure, _AfxCommDlgProc(), which is set up in the CFileDialog (and other common dialog class) constructor.

Because _AfxCommDlgProc() is the common hook procedure for all the common dialogs, you can imagine that it is a pretty long utility function. Listing 6-20 contains an abbreviated version of the _AfxCommDlgProc() utility function, which lives in DLGCOMM.CPP, and highlights the areas that apply to CFileDialog.

Listing 6-20. The _AfxCommDlgProc() MFC common dialog hook utility function

```

UINT CALLBACK _AfxCommDlgProc(HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{
    //Lots of code omitted, including pDlg declaration.
    if (message == nMsgSHAREVI)
        return ((CFileDialog*)pDlg)->OnShareViolation((LPCTSTR)lParam);
    else if (message == nMsgFILEOK) {
        if (afxData.bWin4)
            ((CFileDialog*)pDlg)->m_pofnTemp = (OPENFILENAME*)lParam;
        BOOL bResult = ((CFileDialog*)pDlg)->OnFileNameOK();
        ((CFileDialog*)pDlg)->m_pofnTemp = NULL;
        return bResult;
    }
    else if (message == nMsgLBSELCHANGE){
        ((CFileDialog*)pDlg)->OnLBSelChangedNotify(wParam, LOWORD(lParam),
                                                    HIWORD(lParam));
        return 0;
    }
    else if (message == _afxNMmsgSETRGB)
        return 0;
    return 0;
}

```

_AfxCommDlgProc() checks for different values of the message argument and calls the appropriate virtual callback for that type. For some of the virtual callbacks, like OnShareViolation(), _AfxCommDlgProc() can just call the function and pass through one of its arguments. For other callbacks, like OnFileNameOK(), it has to do some more work.

For OnFileNameOK(), _AfxCommDlgProc() stores the OPENFILENAME that is passed through the lParam argument in the m_pofnTemp argument and then calls OnFileNameOK(). Normally you should be able to look at m_ofn.lpszFileName during OnFileNameOK(), so why would Microsoft store away the OPENFILENAME pointer in m_pofnTemp? It appears that Microsoft did so to work around either a Windows bug or a problem with the Macintosh portability layer.

This sounds plain and simple, but the big question is What is the message “nMsgSHAREVI” and the other values that _AfxCommDlgProc() is checking against here?

If you look at the top of DLGCOMM.CPP, you will find the answer. MFC registers a group of messages from strings defined by the common dialogs. For example, here’s the declaration for the message that specifies that there was a sharing violation:

```
static const UINT nMsgSHAREVI = ::RegisterWindowMessage(SHAREVISTRING);
```

The SHAREVISTRING is a macro that is defined by the Win32 common dialogs. These messages are registered at the start of every MFC program (even if you do not use the common dialogs, incidentally), and _AfxCommDlgProc() uses them to decode the messages sent by the common dialogs and turn them into calls to the common dialog class's virtual callbacks.

But wait, there's still a gnawing question. _AfxCommDlgProc() calls three virtual callbacks: OnShareViolation(), OnFileNameOK(), and OnLBSelChangedNotify(). What about the undocumented virtual callbacks: OnInitDone(), OnFileNameChange(), OnFolderChange(), and OnTypeChange()?

The answer lies in the Windows 95 common dialogs. These dialogs use a different scheme to communicate callbacks to the application. Instead of using the registered message scheme that we have covered, the Windows 95 common dialogs send WM_NOTIFY instead. If the WM_NOTIFY is not handled, then the registered messages are sent.

CFileDialog::OnNotify() Windows 95 Callback Generator

The undocumented CFileDialog::OnNotify() member does the work of turning the Windows 95 WM_NOTIFY messages into the undocumented virtual callbacks. Listing 6-21 shows an abbreviated version of CFileDialog::OnNotify(), from DLGFILE.CPP.

Version Note

This scheme only exists in MFC 4.0 and greater.

Listing 6-21. CDlgFile::OnNotify() pseudocode, from DLGFILE.CPP

```
BOOL CFileDialog::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
    OFNOTIFY* pNotify = (OFNOTIFY*)lParam;
    switch(pNotify->hdr.code)
    {
        case CDN_INITDONE:
            OnInitDone();
            return TRUE;
        case CDN_SELCHANGE:
            OnFileNameChange();
            return TRUE;
        case CDN_FOLDERCHANGE:
            OnFolderChange();
            return TRUE;
        case CDN_SHAREVIOLATION:
            *pResult = OnShareViolation(pNotify->pszFile);
            return TRUE;
    }
}
```

```

    case CDN_FILEOK:
        *pResult = OnFileNameOK();
        return TRUE;
    case CDN_TYPECHANGE:
        OnTypeChange();
        return TRUE;
    }
    return FALSE; // not handled
}

```

First, `OnNotify()` pulls out an `OFNOTIFY` pointer out of the `IParam` argument. The only field of interest is the `pNotify->hdr.code`, which contains the notification code. Next, `OnNotify()` switches off of the notification code and calls the appropriate virtual callback, returning `TRUE` to indicate that it has handled the message.

Here's an interesting question to see if you're on your toes: Why didn't `_AfxCommDlgProc()` just use a switch statement instead of the messy `if/else if` blocks?

Give up? The values checked in `_AfxCommDlgProc()` are not known at compile time, so Microsoft has to use `if/else` blocks instead of a switch statement.

The `CDialog::OnNotify()` approach handles many more notifications than the older `_AfxCommDlgProc()` hook approach, which makes it even easier to customize the Windows 95 common dialog.

The MFC `CFileDialog` class has another undocumented convenience for developers who want to customize the file common dialog. The member function `SetTemplate()` takes two dialog templates, one for the old-style file dialog and one for the new-style one. This undocumented member takes care of determining and using the correct dialog template for you so that you do not have to worry about it in your code! Pretty handy if you still want to support the old-style dialogs.

Now you should have a pretty good understanding of some of the hairier tricks that the common dialogs use. Before we move on, let's take a quick look at one of the member functions that retrieves information from the `CFileDialog`. Let's look at the `GetFileName()` member function, which retrieves the name of the file entered by the user, without path, from the common dialog. Listing 6-22 shows the implementation of the function, which lives in `DLGFILE.CPP`. For example, if we had `C:\SCOT\BOOK\CHAP6.DOC`, the string returned by `GetFileName()` would be "CHAP6.DOC".

Listing 6-22. The `CFileDialog::GetFileName()` implementation, from `DLGFILE.CPP`

```

CString CFileDialog::GetFileName() const
{
    if ((m_ofn.Flags & OFN_EXPLORER) && m_hWnd != NULL){
        CString strResult;

```

```

    if (GetParent()->SendMessage(CDM_GETSPEC, (WPARAM)MAX_PATH,
        (LPARAM)strResult.GetBuffer(MAX_PATH)) < 0)
        strResult.Empty();      '
    else{
        strResult.ReleaseBuffer(); return strResult;
    }
}
return m_ofn.lpstrFileTitle;
}

```

GetFileName() checks if the common dialog is Explorer style (OFN_EXPLORER) and also makes sure it has a valid window handle. If this is the case, it uses a feature of the new common dialogs to send a message (CDM_GETSPEC) to retrieve the file name. If the CDM_GETSPEC message returns anything greater than zero, indicating success, GetFileName() returns the string. If there the SendMessage() fails, or the dialog does not have OFN_EXPLORER set, GetFileName() reverts to the old-fashioned method and returns the m_ofn.lpstrFileTitle field, which is also set by the common dialog.

In the Windows 95 common dialogs, sending messages to the dialog is more reliable; Microsoft encourages Windows 95 developers to switch over to using this technique instead of the old technique of checking the fields of m_ofn.

MFC Common Dialog Wrapup

Now that we've examined the majority of the interesting aspects of the CFileDialog class, it's time to move on to the OLE dialogs. Before we do, here are the key points to remember about the MFC common dialog classes:

- All of the MFC common dialogs derive from CCommonDialog, which provides OnOK() and OnCancel() handling.
- Common dialogs are basically a very thin wrapper around a Win32 structure and API. In our example, these were the OPENFILE structure and the GetOpenFile() / GetSaveFile() APIs.
- MFC takes care of the differences in the old-style and new-style common dialogs by internally checking the OFN_EXPLORER flag and calling the appropriate functions based on the result. This is particularly helpful when it comes to hooks and dialog templates.
- There are some undocumented callbacks that are called through the common dialog hook routines. Before considering using your own hook routine, double-check your version of MFC and see if it is handled but undocumented.

For Further Information

Unfortunately, we can't cover all the MFC common dialogs here. However, if you're interested, here's some food for thought that will get you looking in the right direction.

If you went through the Scribble tutorial, you will recall that the application has a File Open common dialog (among others) from step 1, yet there are no references to CFileDialog in the source code.

Now consider these questions:

- Where does the CFileDialog live?
- If you have a CFileDialog derivative that does something specific for your application, how would you go about making Scribble create your CFileDialog derivative instead of CFileDialog?
- Can you figure out why CFileDialog has the ofnTemp data member?

Take a look at the other common dialog implementations and see if you can discover any undocumented functions and data members. Can you deduce what they are used for? Here are some specific questions to try to answer along the way:

- Why does CFontDialog have an undocumented LOGFONT data member (`m_lf`)?
- Why does CColorDialog need to have an `OnCtlColor` member? What does it do?
- What is `_AFX_COLOR_STATE` and what is it used for?
- What does CPageSetupDialog::PaintHookProc() do and why is it needed?
- What do you think the `AfxCreateDC()` utility function is doing in DLGPRNT.CPP?

There's a gold mine of information buried under the MFC source code. These are just a few questions that will help you dig up some nuggets.

OLE Dialogs

The OLE dynamic link library, OLEDLG.DLL, contains several "OLE common dialogs" that prompt users for certain OLE tasks. For example, when the user inserts a new object, there is a canned dialog that will ask the user to select the type of object from those registered.

Version Note

In MFC versions 3.x and older, the OLE dialog code lived in MFCUIA32.DLL. With Windows 95 and NT 3.51, these functions were merged into the operating system.

By using the common OLE dialogs, like the regular common dialogs, you ensure that end users encounter a similar OLE interface in all of their Windows (and MFC) applications.

Here's a summary of the OLE dialogs and what they do:

- COleDialog—The base class for all OLE dialogs.
- COleInsertDialog—Prompts the user for the type of object to insert; usually created when Edit/Insert Object is selected from the menu.
- COleConvertDialog—Lets the user change the type of an embedded object.
- COleChangeIconDialog—Allows the user to change the icon displayed for a certain type of embedded object.
- COlePasteSpecialDialog—Lets the user select if he or she wants to paste an object, link, or copy into the document. Usually displayed via the Edit/Paste Special menu.
- COleLinksDialog—Allows the user to change OLE links.
- COleUpdateDialog—Lets the user update OLE links.
- COleBusyDialog—Displayed when an OLE server is busy and cannot respond to a request.
- COlePropertiesDialog—Displays property pages for an OLE object.
- COleChangeSourceDialog—Allows the user to change the source or destination of a link.

Using the OLE Dialogs

In most cases, the MFC OLE classes handle the creation of the OLE dialogs for you. MFC OLE dialogs follow the exact same usage as the common dialogs. You create an instance, call DoModal(), and finally retrieve the information the user selected if the DoModal() return value is IDOK.

MFC OLE Dialog Class Internals

The declarations for the OLE dialog classes are in the AFXODLGS.H file. The implementation for the dialogs is spread between three implementation files, which are appropriately named OLEDLGS1.CPP, OLEDLGS2.CPP, and OLEDLGS3.CPP. Here's a guide to which class lives in which source file:

- OLEDLGS1.CPP—COleInsertDialog, COleConvertDialog, COleChangeIconDialog, COleLinksDialog, COleUpdateDialog, COlePasteSpecialDialog

- OLEDLGS2.CPP—COleDialog, COleBusyDialog
- OLEDLGS3.CPP—COlePropertiesDialog, COleChangeSourceDialog

Also, some of the OLE dialog members are implemented in the inline file AFXOLE.INL.

The OLE dialog classes are very similar to the common dialog classes covered earlier in this chapter. Each of the OLE dialogs has a structure that defines the options for the dialog and an API that displays the dialog using the fields set in the structure. For example, the COlePasteSpecialDialog OLE dialog class wraps the OLEUIPASTESPECIAL structure and the OleUIPasteSpecial() OLE API.

The biggest difference between the MFC common dialog classes and the OLE dialog classes is that there is a new base class in the OLE dialog hierarchy between CCommonDialog and each OLE dialog class: it is COleDialog.

Inside COleDialog: The OLE Dialog Base Class

The abbreviated declaration of COleDialog, from AFXODLGS.H, can be found in Listing 6-23.

Listing 6-23. The declaration of the COleDialog OLE dialog base class, from AFXODLGS.H

```
class COleDialog : public CCommonDialog
{
    DECLARE_DYNAMIC(COleDialog)
// Attributes
public:
    UINT GetLastError() const;
// Implementation
public:
    int MapResult(UINT nResult);
    COleDialog(CWnd* pParentWnd);

protected:
    UINT m_nLastError;
protected:
    friend UINT CALLBACK _AfxOleHookProc(HWND, UINT, WPARAM, LPARAM);
};
```

COleDialog is a very small base class that really only adds the m_nLastError UINT, which stores an error code, and the MapResult() member function, which maps some of the OLE return codes to the more MFC-friendly IDOK and IDCANCEL. The GetLastError() member function just returns the contents of m_nLastError.

A hook procedure similar to the good old `_AfxCommDlgProc()`, `_AfxOleHookProc()`, is declared as a friend in the base class so that each derivative does not have to declare `_AfxCommDlgProc()` as a friend.

OLE Dialog Summary

To recap what we discovered about OLE dialogs:

- They are very similar to the common dialog classes, both in use and implementation.
- The OLE dialogs all derive from `COleDialog`, which adds some error handling and a couple of utility member functions.
- MFC already creates a variety of the dialogs in standard situations. It is up to you to handle application-specific situations.

For Further Information

If you are an aspiring OLE guru, here are some questions for you to search out in the MFC source code if you want to learn more:

- What do you think the two utility functions `_AfxOlePropertiesEnabled()` and `_AfxParseDisplayName()` are doing?
- There is an undocumented class, `COleUILinkInfo`. What is it used for? Why do you think it is undocumented? Could you use this undocumented class in your applications?
- Can you deduce what each member of `COlePropertiesDialog` is used for?
- What's the link between `COlePropertyPage` and `COlePropertiesDialog`?

So far we've covered almost all of the `CDialog` derivatives, but if you have an MFC hierarchy chart handy, you will notice that there is still one more `CDialog` derivative: `CPropertyPage`. `CPropertyPage` is a key contributor to the MFC feature known as property sheets, or tabbed dialogs.

Property Sheets (a.k.a. Tabbed Dialogs)

The MFC property sheet dialogs were introduced in MFC 3.0. Figure 6-4 shows an example of a property sheet. Property sheets have become popular lately because they let the end user do a wide variety of operations quickly in one dialog instead of having to traverse several menu/dialog combinations. For an example, check out

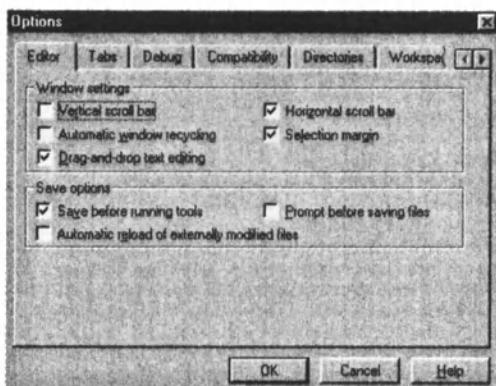


Figure 6-4. An example of a property sheet

the Project/Options tabbed dialog in Visual C++. Imagine how long it would take to try and change several options if each of the tabs were in a separate dialog!

Using MFC Tabbed Dialogs

The MFC implementation of tabbed dialogs is based on two classes, `CPropertySheet` and `CPropertyPage`. `CPropertySheet` is the “tabbed dialog” class, and `CPropertyPage` is a class that encapsulates the “tab,” or page, in the tabbed dialog.

To use the MFC classes, you take the following steps:

1. Create dialog templates that describe the controls and layout that you want for your property sheets.
2. Create a `CPropertyPage` derivative for each template and add data members to store the state of the property page.
- 3a. For a modal property sheet, just create an instance of `CPropertySheet` and call `CPropertySheet::AddPage()` to add the property pages created in the first two steps to the property sheet. After adding the property pages, call `CDialog::DoModal()` to display the property sheet. MFC automatically adds OK/Apply/Cancel buttons for you.
- 3b. If you want a modeless property sheet, you have to create a `CPropertySheet` derivative that adds a way to dismiss the dialog. When you create a modeless `CPropertySheet`, MFC does not add OK/Apply/Cancel buttons for you. Next, create an instance of your `CPropertySheet` derivative and call `CPropertySheet::Create()` to display the modeless property sheet.

Listing 6-24 is an example that creates a property sheet with three tabs. `CPageOne`, `CPageTwo`, and `CPageThree` are `CPropertyPage` derivatives:

Listing 6-24. An example of how to create a modal property sheet with three pages (tabs)

```

//...

CPropertySheet mySheet("My Cool Property Sheet!",this);
// Now create the property pages for the sheet.
CPageOne myPage1;
CPageTwo myPage2;
CPageThree myPage3;

// and insert them into the sheet
mySheet.AddPage(&myPage1);
mySheet.AddPage(&myPage2);
mySheet.AddPage(&myPage3);

// DoModal, gotta love it!
mySheet.DoModal();

//...

```

Moving all the data between your property pages and their member functions could be a drag. But don't worry: you can use DDX and DDV with property sheets. All you have to do is implement a `DoDataExchange()` member in each property page and you're set.

MFC property sheets have one more interesting feature. They let the user apply changes made while the dialog is displayed without causing it to get dismissed. To enable the Apply button, you just have to signal MFC that a value has changed by calling `CPropertyPage::SetModified(TRUE)`. Then when the user presses the Apply button, your member variables will be updated with the new values for the modified page and you can update the user interface accordingly. The "modified" or "dirty" flag is maintained on a per-property page basis, so the entire contents are not applied at once, just the current page, when the user presses the Apply button.

Property sheets are a great tool for MFC programmers. Let's look at how they are implemented and see what's going on inside the MFC property sheet and page classes.

Property Sheet and Page Internals

The two property sheet classes in MFC have a slight identity crisis. The property sheet class is not derived from `CDialog` as you might expect, but the property page class is!

`CPropertySheet` is derived from `CWnd` and not `CDialog`, because `CDialog` is meant to be a Windows dialog encapsulation and Windows dialogs are intimately

tied to dialog template resources. Because CPropertySheets have nothing to do with dialog templates, it makes more sense that they be a generic window, CWnd, derivative and not a CDialog derivative. In fact, it was when CPropertySheet came on the scene that the MFC team pushed many of the CDialog routines (such as RunModalLoop()) up into CWnd from their original home in CDialog. This kept them from having two copies of the same routine: CDialog::RunModalLoop() and CPropertySheet::RunModalLoop()—C++ inheritance at its finest!

This also explains why CPropertyPage derives from CDialog. Because CPropertyPages encapsulate a dialog template, it fits well under CDialog in the MFC hierarchy.

Source Files—and a Word about Versions

You can find the declaration for CPropertySheet and CPropertyPage in AFXDLGS.H. The implementation for both classes is in DLGPROP.CPP, with some inline functions living in AFXDLGS.INL.

With MFC 4.0, the CPropertySheet and CPropertyPage internals changed dramatically. The original MFC 3.0 implementation of property sheets was completely contained inside of MFC. MFC performed all of the drawing of the tabs, window creation, tab management, and so on. At that time, the Windows common controls (a set of new controls that stem from Windows 95) had not been released, so the MFC team went ahead and implemented their own property sheets inside of MFC.

In MFC 4.0, the first major revision after the release of Windows 95, the MFC team completely rewrote the property sheets to use the Win32 property sheet implementation (which uses the common tab control) from the Windows common controls. Now, instead of MFC having to manage every aspect of property sheets, the Windows common controls handle much of the work. MFC just provides to the new Windows common controls the same interface that it has since version 3.0. In addition to the interface we have discussed, the MFC 4.0 version adds a “Wizard” mode, which lets you create wizards using the familiar MFC property sheet classes. This functionality comes from the encapsulated Windows common control.

MFC property sheets are not the only MFC class to undergo this rewrite. Some of the classes we will cover in Chapter 9 (such as toolbars and status bars) have also been rewritten to use the new Windows common controls.

To complicate matters, there are also MFC classes that provide a thin encapsulation over the Windows common controls, including one for the property sheet, CTabCtrl. We’ll cover the CTabCtrl class later, but in most instances it is much easier to use the CPropertySheet/CPropertyPage MFC interfaces for property sheets instead of the much lower level and harder to use CTabCtrl.

When the MFC team rewrote CPropertySheet/CPropertyPage to use the Windows common controls, the amount of code decreased dramatically. If you have an older

version of MFC, open up AFXDLGS.H and compare the declaration of CPropertySheet to the one in MFC 4.0 or greater. There are about one-third the number of members now! But don't worry: some pretty interesting things are still going on.

A Review of the Windows Property Sheet Common Control

Now that the MFC property sheets wrap the property sheet Windows common control, to understand the MFC CPropertySheet/CPropertyPage classes, we need to review how to use the property sheet control.

Using the Windows property sheet common control is very similar to using the common dialogs. First, you fill in fields of a structure, PROPSHEETHEADER, that define the settings you want to make for the property sheet, including the property page information. Once you have the structure completely filled in, you call the Win32 API PropertyPage(), which takes the structure and creates the property sheet.

The PROPSHEETHEADER structure is pretty complicated, but there are a couple of fields that are important to understand the MFC property page classes. One important field is ppsp. This field contains a pointer to an array of property page structures, PROPSHEETPAGE. Each PROPSHEETPAGE structure defines a page, or tab, in the property sheet.

The PROPSHEETHEADER field nPages is used to tell the control how many page structures to expect in this array.

One of the most important fields of the structure, dwFlags, lets you set a variety of different flags to put the property sheet into different modes of operation. The property sheet common control can display tabs in two different ways, stacked or scrolled. Stacked tabs appear to be on top of each other and they are all always visible. Scrolled tabs are all next to each other in the property sheet; a little scroll bar lets you scroll them on and off screen as needed.

The property sheet flags are prefixed with "PSH." Here's a list of some of the flags used by the MFC property sheet classes:

- PSH_MODELESS—Causes the property sheet to be modeless instead of modal.
- PSH_PROPSHEETPAGE—Tells the property sheet that there are property page structures. It will ignore them unless this flag is set.
- PSH_MULTILINETABS—Displays stacked tabs instead of scrolled.
- PSH_WIZARD—Puts the property sheet control into Wizard mode. In Wizard mode, the control does not display tabs but displays Next and Previous buttons that let the user "flip" between pages of the wizard.

After creating the property sheet via the PropertySheet() API, the property sheet sends your application WM_NOTIFY messages as the user interacts with it. These notifications are prefixed with "PSN." Here are some examples:

- PSN_APPLY—The Apply button was pressed.
- PSN_WIZBACK—In Wizard mode, the user pressed the Previous button.
- PSN_WIZNEXT—In Wizard mode, the user pressed the Next button.
- PSN_WIZFINISH—In Wizard mode, the user pressed the Finish button.

You can also send messages to the property sheet control if you want to retrieve information (usually in response to one of the notifications just mentioned). These messages are prefixed with “PSM.” Some examples are these:

- PSM_SETTITLE—Sets the title of the property sheet.
- PSM_ADDPAGE—Adds a page to the property sheet. The IParam of this message is a handle to a property page created with the CreatePropertySheetPage() function.
- PSM_REMOVEPAGE—Deletes a page from a property sheet.

This brief overview should be enough to illustrate what the CPropertySheet/CPropertyPage classes are doing. If you want to customize the MFC property sheets, or learn more about the Windows common controls, look for the complete on-line information as well as the several books on the subject.

Now, let's roll up our sleeves and dig into the MFC property sheet classes. We'll take them top-down, first looking at CPropertySheet and then taking a gander at CPropertyPage.

Inside CPropertySheet

CPropertySheet is a CWnd derivative whose job is to provide a CDialog-like interface to the property sheet common control. As we look inside CPropertySheet, remember to keep an eye out for the hoops MFC has to jump through to map the old MFC 3.0 CPropertySheet interface onto the new Windows common control.

Listing 6-25 contains a very abbreviated version of the CPropertySheet declaration, from AFXDLGS.H.

Listing 6-25. The abbreviated CPropertySheet declaration, from AFXDLGS.H

```
class CPropertySheet : public CWnd
{
// Construction  **omitted
// Attributes   ** mostly omitted
public:
    PROPSHEETHEADER m_psh;
// Operations   ** omitted
public:
```

```

// Implementation  **some omitted
public:
    void CommonConstruct(CWnd* pParentWnd, UINT iSelectPage);
    void EnableStackedTabs(BOOL bStacked);
    virtual void BuildPropPageArray();
    virtual BOOL OnInitDialog();

protected:
    CPtrArray m_pages;      // array of CPropertyPage pointers
    CString m_strCaption;  // caption of the pseudo-dialog
    CWnd* m_pParentWnd;    // parent window of property sheet
    BOOL m_bStacked;        // EnableStackedTabs sets this
    BOOL m_bModeless;       // TRUE when Create called instead of DoModal

// Generated message map functions  **omitted
friend class CPropertyPage;
};


```

As usual, we've omitted just about everything from the declaration except for our favorite section, // Implementation.

Notice that in the public // Attributes section, there is a PROPSHEETHEADER called m_psh. You should be experiencing déjà vu if you just read the CFileDialog description. CFileDialog and the other common dialogs use the exact same approach of maintaining a public member, so you can access and change the dialog/control structure as needed in your MFC applications. (Some of you may be experiencing “vu jàdé”—the odd feeling that you have never heard of this before.)

Moving right along to the // Implementation section, there are lots of member functions and data members, which are briefly described here:

- **CommonConstruct()**—CPropertySheet has tons of constructors. They all massage their arguments and then pass them to CommonConstruct() so that the constructor code is not duplicated for each constructor. CommonConstruct() first sets the dwFlags field of the m_psh structure to PSH_PROPSHEETPAGE and then initializes several other m_psh structure fields. CommonConstruct() also sets m_bStacked to TRUE and m_bModeless to FALSE.
- **EnableStackedTabs()**—Puts the CPropertySheet into stacked versus scrolled tab mode. This inline member function just toggles the m_bStacked member.
- **BuildPropPageArray()**—We'll look at this member function soon. We'll leave it descriptionless for now to build your anticipation.
- **OnInitDialog()**—A very long member that performs these steps:
 1. If not in Wizard mode, it lays out the entire property sheet so that the controls are not bunched together.
 2. Changes the tab style based on the m_bStacked member. It does this via EnableStackedTabs().

3. If the property sheet is modeless and not a wizard, OnInitDialog() removes the standard buttons (remember that the MFC property sheets don't have OK/Cancel/Apply buttons).
 4. Centers the property sheet using CWnd::CenterWindow().
- m_pages—An array of pointers, CPtrArray, that stores pointers to the CPropertyPage page classes for the property sheet.
 - m_strCaption—The caption for the property sheet.
 - m_pParentWnd—A pointer to the parent window.
 - m_bStacked—A Boolean that tracks if the class user has specified stacked or scrolled tabs.
 - m_bModeless—A Boolean that tracks if the class user wants the property sheet to be modal or modeless.

CPropertySheet::DoModal()

To see how all of these members fit together, let's look at CPropertySheet::DoModal(). Listing 6-26 contains a pseudocode version of CPropertySheet::DoModal(), from DLGPROP.CPP.

Listing 6-26. The CPropertySheet::DoModal() pseudocode, from DLGPROP.CPP

```
int CPropertySheet::DoModal()
{
    // register common controls
    VERIFY(AfxDeferRegisterClass(AFX_WNDCOMMCTL_REG));
    // finish building PROPSHEETHEADER structure
    BuildPropPageArray();
    pParentWnd->EnableWindow(FALSE);
    HWND hWnd = (HWND)::PropertySheet(&m_psh);
    nResult = RunModalLoop(dwFlags);
    // cleanup
    DestroyWindow();
    return nResult;
}
```

First, DoModal() makes a call to the mysterious AfxDeferRegisterClass(). It causes MFC to call the InitCommonControls() routine, which initializes the Windows common controls DLL.

Next, DoModal() calls BuildPropPageArray(). We'll cover this soon—hold on. (More anticipation!)

After the BuildPropPageArray() call, DoModal() does the same things as the CDialog and CFileDialog versions. It disables the main window to enforce modality, then calls the ::PropertySheet() API to create and display the common control

property sheet. DoModal() then calls CWnd::RunModalLoop() to handle pumping messages during modality. When RunModalLoop() is done, DoModal() destroys the window, re-enables the parent window, and returns the result that was returned by RunModalLoop(). We have omitted some extra details to the RunModalLoop() call for brevity: check the source code for more details.

One key point to remember about CPropertySheet is that until DoModal() (for modal) or Create() (for modeless) is called, the class does not map to a Windows window handle. The Windows window handle is created and attached to the MFC CPropertySheet object after the ::PropertySheet() API call. As we continue to look at CPropertySheet members, you will notice that they check that m_hWnd != NULL. This means that some of these operations behave differently when called while the dialog is created versus when it is not attached to a window handle. Usually this is more of an issue for modeless property sheets, because they have a longer life than modal property sheets.

CPropertySheet::AddPage()

Let's dissect the AddPage() member function to see how CPropertySheet goes about adding a page. Listing 6-27 contains the pseudocode for CPropertySheet::AddPage() from DLGPROP.CPP.

Listing 6-27. The CPropertySheet::AddPage() pseudocode, from DLGPROP.CPP

```
void CPropertySheet::AddPage(CPropertyPage* pPage)
{
    m_pages.Add(pPage);
    if (m_hWnd != NULL) {
        HPROPSHEETPAGE hPSP = CreatePropertySheetPage(&pPage->m_psp);
        if (!SendMessage(PSM_ADDPAGE, 0, (LPARAM)hPSP)){
            DestroyPropertySheetPage(hPSP);
            AfxThrowMemoryException();
        }
    }
}
```

First, AddPage() stores the CPropertyPage pointer in its internal m_pages property page pointer array. Next, after checking that m_hWnd is non-NULL, AddPage() calls ::CreatePropertySheetPage() and sends the property sheet control the PSM_ADDPAGE message to get it to add the page returned by CreatePropertySheetPage.

You might be wondering why CPropertySheet maintains the m_pages array when it could just use the property pages that are stored in the control. Well, it's a matter of timing. In most cases the m_hWnd != NULL check will fail, because usually MFC users call AddPage() for each property page and then DoModal() or Create().

The `m_pages` array buffers up all of the property pages added before property sheet creation.

As you may have guessed, `BuildPropPageArray()` takes care of converting the `m_pages` array into the `PROPSHEETPAGE` structures needed by the `m_psh` `PROPSHEETHEADER` structure. Let's take a look at `BuildPropPageArray()` and see what is involved in this conversion.

The Wait Is Over: `CPropertySheet::BuildPropPageArray()`

Listing 6-28 shows the `BuildPropPageArray()` pseudocode, from `DLGPROP.CPP`

Listing 6-28. The `CPropertySheet::BuildPropPageArray()` pseudocode, from `DLGPROP.CPP`

```
void CPropertySheet::BuildPropPageArray()
{
    delete[] (PROPSHEETPAGE*)m_psh.ppsp;
    m_psh.ppsp = NULL;
    LPPROPSHEETPAGE ppsp = new PROPSHEETPAGE[m_pages.GetSize()];
    m_psh.ppsp = ppsp;
    for (int i = 0; i < m_pages.GetSize(); i++) {
        CPropertyPage* pPage = (CPropertyPage*)m_pages[i];
        memcpy(&ppsp[i], &pPage->m_psp, sizeof(PROPSHEETPAGE));
        // **omitted - find/load/lock ppsp[i] resource
        // of psp.pszTemplate into pTemplate
        _ChangePropPageFont(pTemplate);
    }
    m_psh.nPages = m_pages.GetSize();
}
```

First, `BuildPropPageArray()` deletes and nulls out the `m_psh.ppsp` field to make room for a new array that is built from the internal `CPtrArray`, `m_pages`.

Next, `BuildPropPageArray()` allocates a new array of `PROPSHEETPAGES` into the local `ppsp` variable and sets `m_psh.ppsp` to point to that same block of memory. They do this so that they don't have to access the `m_psh` structure every time they want to change the `.ppsp` field: they can do so through the local `ppsp` variable instead.

After setting up the the `ppsp` field memory, `BuildPropPageArray()` iterates through the `CPropertyPages` in the `m_pages` array, and for each page it copies the `CPropertyPage`'s internal `PROPSHEETPAGE` structure into the `PROPSHEETHEADER` `ppsp` array. After copying the page, `BuildPropPageArray()` calls `_ChangePropPageFont()` with a pointer to the dialog template to change the fonts of all of the controls so that they match the font used by the property sheet.

Once all of the pages have been put into `m_psh.ppsp`, `BuildPropPageArray()` updates the page count in `m_psh.nPages` to match the size of `m_pages`.

CPropertySheet Notifications

CPropertySheet maps several of the property sheet common control notifications into overrideable callbacks. For example, when in Wizard mode, CPropertySheet will receive WM_NOTIFY messages in its OnNotify() handler, such as PSN_WIZNEXT. When it gets the call, it calls the virtual OnWizardNext() member function to signal that the Next button has been called.

CPropertyPage

CPropertyPage, like CPropertySheet, encapsulates a structure, PROPSHEETPAGE. The PROPSHEETPAGE structure is pretty straightforward. CPropertyPage is a very thin wrapper around the structure: it adds a good bit of logic on top of PROPSHEETHEADER.

Since this is the case, we'll leave a detailed exploration of CPropertyPage up to you. Like CPropertySheet, CPropertyPage's declaration is in AFXDLGS.H and its implementation lives in DLGPROP.CPP.

Property Sheet Recap

The basic things we learned in our MFC property sheet travels are these:

- The MFC property sheet classes provide a CDlg interface to the property sheet common control. In doing so, they have to sometimes perform some pretty weird conversions.
- Like the common dialogs, these classes encapsulate a Win32 structure and API. The structure is public and can be manipulated by any class user. Using this knowledge, you pretty much have free rein to set the different flags in the structure as needed by your application.

For More Information

For you die-hard property sheet lovers, here are some questions that will lead you to some interesting areas not covered here:

- Explore AfxDeferRegisterClass().
- How does _ChangePropPageFont(), a static function, know what MFC is going to use as the property sheet font?
- CPropertySheet has a member function, GetTabControl(), that returns a MFC control class pointer, CTabCtrl, to the underlying Windows property sheet common control used by CPropertySheet.
 1. How does GetTabControl() go about doing this?
 2. Why do you think they do not use CTabCtrl inside CPropertySheet instead of calling the APIs directly?

- Look at a pre-MFC 4.0 version of CPropertyPage and CPropertySheet.
 1. Can you locate where and how this version draws itself?
 2. Are there any undocumented helper classes?
 3. Which do you think is cleaner, pre-MFC 4.0 or post-MFC 4.0?

Now we are officially done exploring the MFC CDIALOG derivatives. As advertised, it's time to move on to the MFC control classes.

MFC Control Classes

With the introduction of the Windows common control classes into MFC, there are now three groups of MFC control classes.

1. The “old fashioned” classes that encapsulate those six classic controls which have been in Windows since the beginning: the button, combo box, edit, list box, scroll bar, and static. In MFC 4.0, Microsoft introduced two new extensions of these classics: a drag-and-drop list (which is actually one of the common controls) and a check list box. MFC has also had a bitmap button class since version 2.0.
2. The “new-fangled” classes that encapsulate the Windows common controls.
3. The OLE controls. OLE controls have enough interesting internals that they have their own chapter, Chapter 15. We won’t be covering them here.

In this section, we’ll look first at the classic control classes (wow, that has a nice ring to it!) and then take a look inside the newer common control classes.

The “Old-Fashioned” Windows Control Classes

The classic control classes are these:

CBitmap
CBitmapButton
CComboBox
CEdit
CListBox
CDragListBox
CCheckListBox

CScrollBar

CStatic

The names of these classes are pretty self-explanatory. One way to use one of the MFC classic control classes is to create it on the fly from an existing control in a dialog:

```
CButton * pButton = (CButton *)pDialog->GetDlgItem(IDC_CHECK1);
ASSERT(pButton != NULL);
pButton->SetCheck(m_bShowState);
```

Another way to use these classes is to make one explicitly by creating the object and then calling the Create() member function. For example, this code creates a button in a CView:

```
CRect rectButton(10,10,50,50);
CButton * pButton = new CButton;
pButton->Create("Button text", WS_CHILD|BS_PUSHBUTTON,
rectButton, pView, ID_BUTTON);
// Later that same day...
pButton->DestroyWindow();
delete pButton;
```

After you have created the control, there are a variety of member functions that you can call that are specific to each control. Check the MFC documentation for details.

Inside the Classic Control Classes

The MFC control classes are all CWnd derivatives, and their declarations live in the AFXWIN.H MFC header file. The implementations are spread between several files:

- AFXWIN2.INL—Inline members of all control classes.
- WINCTRL1.CPP—CStatic, CButton, CListBox, CComboBox, CEdit, and CScrollBar.
- WINCTRL2.CPP—CDragListBox.
- WINCTRL3.CPP—CCheckListBox.
- WINBTN.CPP—CBitmapButton.

The implementation of most of the “basic” six controls is pretty, well, bland. When the Create() member function is created, MFC attaches itself to the corresponding Windows object. From there all of the member functions map to the messages

supported by each control. For example, when you add a string to a CListBox via CListBox::AddString(), the following code from AFXWIN2.INL is called:

```
ASSERT(::IsWindow(m_hWnd));
return (int)::SendMessage(m_hWnd, LB_ADDSTRING, 0, LPARAM)lpszItem);
```

That's about all there is to it! In fact, if you open up WINCTRL1.CPP and look at the // Implementation section for these classes, you'll see that there just isn't anything there!

Don't despair: if we look a little further, we will find some interesting internals to explore. The implementation of the CCheckListBox is pretty interesting, plus it's a great example of how to customize one of the existing control classes. Let's check it out.

Before we look at the declaration, a little introduction: a check list box is a list of checkboxes. These controls are handy if you have a list of options (you see them used in most Install programs). Using CCheckListBox is very similar to using a CListBox, except there are some new member functions for dealing with the checkbox states. For example, there is a SetCheck() member function for setting the check on an item, and there is a GetCheck() member function for retrieving the check.

You can also specify one of the button styles to apply to all of the buttons in the check list box: BS_CHECKBOX, BS_AUTOCHECKBOX, BS_AUTO3STATE, and BS_3STATE.

Exploring CCheckListBox

Listing 6-29 contains a very abbreviated declaration for CCheckListBox, from AFXWIN.H.

Listing 6-29. The abbreviated CCheckListBox declaration, from AFXWIN.H

```
class CCheckListBox : public CListBox
{
// Constructors **omitted
// Attributes **omitted
// Overrideables **omitted
// Implementation
protected:
    void PreDrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
    void PreMeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
    int PreCompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
    void PreDeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct);
    virtual BOOL OnChildNotify(UINT, WPARAM, LPARAM, LRESULT*);
    int CalcMinimumItemHeight();
    void InvalidateCheck(int nIndex);
```

```

void InvalidateItem(int nIndex);
int m_cyText;
UINT m_nStyle;
// Message map functions **some omitted
protected:
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg LRESULT OnLBSetItemData(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnLBSetItemHeight(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
};

}

```

CCheckList uses the owner-draw feature of the standard list box class to implement the list of checkboxes. The first five members, PreXXX, all take part in answering various messages that are sent by a list box that is in owner-draw mode. **OnChildNotify()** takes care of mapping the owner-draw messages to these member functions.

Next are some utility functions. **CalcMinimumItemHeight()** uses font metrics to figure out the minimum size to use for each item in the list. **InvalidateCheck()** and **InvalidateItem()** are utility members that invalidate either the check area of an item or the entire item.

CCheckListBox has two data members. The **m_cyText** data member stores the height of the text in the checklist, and **m_nStyle** holds the button style for the check list box.

CCheckListBox has a message map that it uses to subclass many of the standard **CListBox** messages. For example, **OnLButtonDown()** will now need to figure out if the user is trying to check one of the checkboxes and update the state of the button. Normally, **OnLButtonDown** would just select an entry in the list box.

Let's jump right into the **CCheckListBox** implementation, as shown in Listing 6-30, and look at the **CCheckListBox::SetCheck** member to see where the information about checks is stored. The implementation lives in **WINCTRL3.CPP**.

Listing 6-30. The abbreviated **CCheckListBox::SetCheck() implementation, from **WINCTRL3.CPP****

```

void CCheckListBox::SetCheck(int nIndex, int nCheck)
{
    LRESULT lResult = DefWindowProc(LB_GETITEMDATA, nIndex, 0);
    AFX_CHECK_DATA* pState = (AFX_CHECK_DATA*)lResult;
    if (pState == NULL)
        pState = new AFX_CHECK_DATA;
    pState->m_nCheck = nCheck;
    VERIFY(DefWindowProc(LB_SETITEMDATA, nIndex, (LPARAM)pState) != LB_ERR);
    InvalidateCheck(nIndex);
}

```

First, SetCheck() calls DefWindowProc() with LB_GETITEMDATA to see if an AFX_CHECK_DATA pointer has already been created for the item (more on this in a minute). If not, SetCheck() goes ahead and allocates one.

Next, SetCheck() changes the m_nCheck member of the AFX_CHECK_DATA state pointer and re-associates the new pointer using DefWindowProc(LB_SETITEMDATA).

Finally, SetCheck() calls InvalidateCheck(), which will invalidate the check, which in turn generates a WM_DRAWITEM call. The PreDrawItem() member function will then draw the item to reflect the new nCheck state.

The AFX_CHECK_DATA seems to be keeping the state for each item. Here's its declaration, from WINCTRL3.CPP:

```
struct AFX_CHECK_DATA
{
public:
    int m_nCheck;
    BOOL m_bEnabled;
    DWORD m_dwUserData;
};
```

So, for each item (checkbox) in the checklist, CCheckList maintains the check state (m_nCheck), if it is enabled (m_bEnabled), and also a DWORD, so that the class user can still attach some data to the item (m_dwUserData).

The CCheckList::PreDrawItem() member function is the key to the entire class. It does all of the drawing for CCheckList. Unfortunately, this function is too long to reproduce here, even using pseudocode. Here's a synopsis of what it does (you may get more from this if you open up WINCTRL3.CPP and read the code as you read the description):

1. Gets a pointer to a static _AFX_CHECKLIST_STATE called pChecklistState (more on this later).
2. Determines if it needs to draw the entire checkbox (check and text) or a portion (check or text) based on the exposed region.
3. If the check needs to be drawn, it creates a compatible DC and selects a bitmap from the pChecklistState pointer and uses it to BitBlt the check.
4. Now PreDrawItem calls DrawItem.

CCheckListBox::DrawItem() goes on from there and uses ExtTextOut() to draw the text of the checkbox item after selecting in the appropriate fonts and other DC objects.

One interesting technique employed by this CListBox derivative is the use of that AFX_CHECKLIST_STATE. Let's take a closer look and then move on.

The declaration of `_AFX_CHECKLIST_STATE`, from WINCTRL3.CPP is this:

```
class _AFX_CHECKLIST_STATE : public CNoTrackObject
{
public:
    _AFX_CHECKLIST_STATE();
    virtual ~_AFX_CHECKLIST_STATE();
    HBITMAP m_hbitmapCheck;
    CSize m_sizeCheck;
};
```

By deriving from `CNoTrackObject`, you tell MFC that you do not want your class to be tracked by the memory diagnostics. Since this is a static object, it will not get destroyed until the program exits, so MFC would erroneously report it as a memory leak.

You'll notice that the `m_hbitmapCheck` bitmap handle data member that we saw being used in `PreDrawItem()` is here, along with `m_sizeCheck`, which keeps the size of the bitmap. Here's an excerpt from the `_AFX_CHECKLIST_STATE` constructor that shows how it gets this handle and the size of the bitmap.

```
CBitmap bitmap;
if (afxData.bWin4 || AfxGetCtl3dState()->m_pfnSubclassDlgEx != NULL)
    VERIFY(bitmap.LoadBitmap(AFX_IDB_CHECKLISTBOX_95));
else
    VERIFY(bitmap.LoadBitmap(AFX_IDB_CHECKLISTBOX_NT));
BITMAP bm;
bitmap.GetObject(sizeof(BITMAP), &bm);
m_sizeCheck.cx = bm.bmWidth / 3;
m_sizeCheck.cy = bm.bmHeight;
m_hbitmapCheck = (HBITMAP)bitmap.Detach();
```

The `_AFX_CHECKLIST_STATE` constructor loads a different bitmap based on if it is running on Windows 95 or not. Once it loads the bitmap from resource, it uses `GetObject` to determine the width and height, which it stores in `m_sizeCheck`. Finally, it stores the handle to the bitmap returned by `CBitmap::Detach()` in `m_hbitmapCheck`.

These bitmaps are `MFC\INCLUDE\RES\NTCHECK.BMP` and `MFC\INCLUDE\RES\95CHECK.BMP`. Figures 6-5 and 6-6 show the two bitmaps that are used to display all of the different check types. Each segment represents a state. You will see the last state only in a three-state (AUTO_3STATE) checkbox. `PreDrawItem` takes care of chopping out the correct bitmap based on the state. For fun, take a look at the `PreDrawItem()` source and see how it does this.



Figure 6-5. NT version of the check bitmaps



Figure 6-6. Windows 95 version of the check bitmaps

The “New-Fangled” MFC Windows Common Control Classes

Like the classic control classes, most of the common control classes are derived from CWnd. Here’s a list of the common control classes, along with a brief description. If the class is not derived from CWnd, the base class is noted.

- CAnimateCtrl—A control that plays animations.
- CDragListBox—A CListBox derivative that lets you drag and drop items in a list box.
- CHeaderCtrl—Used with a CListCtrl to display columnar information.
- CHotKeyCtrl—Provides an interface for getting key sequences from the user (Alt-Backspace-Delete).
- CImageList—A CObject derivative that maintains a collection of images for you.
- CListCtrl—Displays a graphical list (Explorer-like) of list items.
- CProgressCtrl—Displays a progress bar.
- CRichEditCtrl—A rich edit control that understands some basic RTF formatting and allows multiple fonts, colors, and more..
- CSliderCtrl—A slider for choosing between a range of values.
- CSpinButtonCtrl—A spinner.
- CStatusBarCtrl—A status bar.
- CTabCtrl—The property sheet control.
- CToolBarCtrl—Implements a toolbar.
- CToolTipCtrl—Provides tool tips.
- CTreeCtrl—An Explorer-like tree control.

Using the Common Control Classes

Using the MFC common control classes is no different from using the classic control classes. You either put the control in a dialog using the resource editor or use it as a child window using ::Create to create the control and DestroyWindow to nuke it.

The major difference is of course the variety of new member functions that are available to you. Taking a quick look at the CTreeCtrl, there are easily 30 or 40 member functions that allow you to manipulate the tree control.

Inside the MFC Common Control Classes

The declarations for the common control classes are in the AFXCMN.H header file. The implementations are spread between these files:

- AFXCMN.INL—Lots of inline members for the common control classes.
- WINCTRL2.CPP—All but CRichEditCtrl.
- WINCTRL4.CPP—CRichEditCtrl.

The common control classes are a very thin wrapper around their corresponding controls. Like the classic control classes, they attach at create, and from there each member function basically sends a message to the control to either set or retrieve information.

For example, the CTreeCtrl::InsertItem() member function lives in AFXCMN.INL and expands to this:

```
HTREEITEM CTreeCtrl::InsertItem(LPTV_INSERTSTRUCT lpInsertStruct)
{
    ASSERT(::IsWindow(m_hWnd));
    return (HTREEITEM)::SendMessage(m_hWnd, TVM_INSERTITEM, 0,
        (LPARAM)lpInsertStruct);
}
```

So, other than adding some asserts and massaging the occasional argument or two, there really aren't any exciting internals to cover for the common control classes.

It is still worth your time to quickly browse through the files AFXCMN.H, AFXCMN.INL, WINCTRL2.CPP, and WINCTRL4.CPP to get a feel for how MFC encapsulates the controls. Someday you might want to send your own messages or grab certain notifications from the controls, and having an understanding of how MFC does this will be helpful.

Conclusion

In the next chapter we will continue climbing up the MFC tree and explore the internals of the document/view architecture. Because this is entirely implemented inside MFC, there are lots of ripe internals ready for the picking.

MFC's Document/ View Architecture

As an application framework, MFC is fairly rich and powerful. In addition to providing a convenient wrapper for the Windows SDK, MFC offers several features that make your life as a developer quite a bit simpler. These consist of such elements as easy-to-implement toolbars and status bars, logical command and message routing, prebuilt collection classes, and a string class replete with assignment and concatenation operators. One of MFC's most important features is its ability to separate code that manages an application's data from code that renders that data. This specific feature—the separation of an application's data management code from its user-interface code—is provided by MFC's document/view architecture.

The document/view architecture is one of the cornerstones of MFC, and understanding it is critical to using MFC effectively. For example, MFC relies heavily on the document/view architecture to implement OLE compound documents, and you must have a firm grasp of that architecture to implement compound documents using MFC.

Why You Want Document/View

Almost every software development effort has had to deal with data management in some form or other. Figuring out ways of managing an application's data has been a problem for software developers through the history of our industry. After all, information and data management is one of the primary uses for computer technology.

There are many issues involved in separating data from user-interface display code, including (1) deciding which part of the application owns the data, (2) deciding which part of the application is responsible for updating the data, (3) figuring out how to display multiple renderings of the data, (4) coordinating data updates, (5) conceiving of some way to store the data, and (6) managing the user interface.

The last item can be tricky—especially when multiple document types are involved, because that often requires updating toolbars and menus.

If you come from a C/SDK background, you probably understand how difficult it can be to manage data. Because of the structure of an SDK program, there's no easy way to decide where to put the data management code.

Other Reasons

Separating data from GUI code is certainly an important goal, but there are even more benefits to using MFC's document/view architecture. One major reward is the built-in printing and print preview support. Who can deny the power you feel when you write your first MFC application using documents and views: in a matter of hours your application's printing and print preview support rivals that of most commercial Windows applications.

As you write more-advanced applications, you may find that you need to support OLE features such as compound documents and in-place editing. MFC's support for these OLE features depends heavily on the document/view architecture.

The Old Way

Remember in the old days, when you had to write your application using C and the SDK (or even using MFC version 1—before Microsoft added all the high-level abstractions)? You had to find some way of representing your data. Then you had to figure out some way of rendering the data on the screen. If there was more than one way of representing the data, it was your responsibility to coordinate the application data and the different renderings of that data. It was a difficult process, and there were as many ways to implement the data management as there were programmers. From the C/SDK perspective, there are several approaches you can take to solve these problems.

If your application uses only one kind of data (for instance, a simple notepad-type editor that edits lines of text), this problem isn't too bad. For such an uncomplicated task, a global data definition suffices. For a simple text editor, the data is perhaps an array of strings representing the text.

But what if you want to write an application that uses multiple types of data? One example might be an application that manages several different types of databases. It's conceivable that each type of document requires its own set of menu resources and toolbars. In addition to writing code to manage the databases, you have to write

the code to manage the user interface for each of the different document types and their resources.

Next imagine that you want to create an application that renders its data in several different ways. Perhaps the application edits a spreadsheet or database that displays its data as tables as well as various types of graphs. Further imagine that you can actually modify the data from each of the renderings and that a change in one rendering causes the other renderings to be updated. Now you need to develop software that manages data display as well as coordinates user-interface interactions and manages the application data.

You can quickly see how the problems begin to mount up. Fortunately, there is a common way to handle all these issues: it's called the document/view architecture.

The basic idea behind any data management scheme is to split data management into two conceptual parts: (1) data management and (2) user-interface(s) management. This is just what the document/view architecture does.

The Architecture

Supporting the coordination of application data and data renderings is a perfect job for a framework. And when Microsoft released MFC 2.0 in early 1993, it included a document/view architecture within MFC. MFC's document/view architecture deals with the data management and user-interface issues just described.

As an aside, MFC's document/view is not a new idea. It was first created by the folks at Xerox PARC (by the same company that invented graphical user interfaces) and was a key part of the Smalltalk environment. Smalltalk's version of document/view is called model-view-controller (MVC) (where model=document). The Smalltalk MVC architecture has a controller (kind of like CDocTemplate in MFC) that acts like a shield between the document and the view so that they do not get too dependent on each other.

Documents and Views

The MFC document/view architecture provides a single, consistent way of coordinating application data and renderings of that data. In MFC, the document handles data management and an application's views handle the user-interface management. In effect, an application's data is centralized in one place, and the user-interface code is packaged separately.

The main advantage of the document/view architecture is that it encourages a clear separation between application data (documents) and representations of that data (views). The term *document* is a little misleading, because it seems to imply

data in the form of things like word processors and spreadsheets. A better term might be *data set* or *data source*. A document is just a place to keep data, and a view is a place to represent that data.

This separation is important—especially if you have multiple ways to represent your data. It would be a great deal of work to write the code to output the data mixed in with the code that manages the data. Though you could write your own code to accomplish this separation, the document/view architecture gives you a single, consistent way to do it. The framework provides the basic infrastructure for managing documents and views, including such functions as file management, drawing on the screen, and managing resources. You just customize (or ignore) the parts that don't meet your needs.

In the absence of a framework to take care of this separation, you'd need to deal with a variety of issues. For instance, if you have more than one way to look at the data, you need to figure out a way to make sure that the current representation (view) accesses the current data properly. And if the user can edit data from one or more representations, you must make sure that all the representations are updated. MFC's document/view architecture does all this work already!

Document/View Components

There are four main components to MFC's document/view architecture: (1) documents, (2) views, (3) document/view frames, and (4) document templates. The following sections explain each component.

Documents

At the core of MFC's document/view architecture is the document itself, which you can think of as any data source.

For example, if you're developing an application for monitoring real-time data collection, there's no reason why the document can't be the real-time data supplier itself. Naturally, document management tends to involve reading and writing to and from persistent storage (like a hard disk), but it's not absolutely necessary.

In MFC, the document is embodied by the CDocument class. When writing your own MFC application, you should derive your document from CDocument. In fact, when you use AppWizard to generate an application, AppWizard generates a class derived from CDocument. CDocument has various functions for performing such operations as managing file I/O and updating renderings of the data.

The CDocument class is fairly useless by itself. When you implement a document within your MFC application, you usually add data members to your CDocument class to represent the data within the document. For instance, if you're writing a spreadsheet-type application, you might create a document class that handles an array

of structures representing spreadsheet cells. If you're developing a drawing application, you might represent your data as an array or list of structures describing various drawings on the screen. Of course, the classic example of a document is found in Microsoft's own MFC primer example: the Scribble application. Microsoft's Scribble application represents its data as an array of positions (coordinates) on the screen.

MFC's CDocument class is two layers deep in the MFC hierarchy. It's derived from CCmdTarget, which is derived from CObject. As a result of being derived from CObject, CDocument is eligible for all the support offered by the CObject class. That is, run-time type information, dynamic creation, and serialization. In addition, by virtue of being derived from CCmdTarget, the CDocument class can receive command messages (that is, WM_COMMAND messages). Finally, CDocument can sport Component Object Model (COM) interfaces, as well as OLE automation, because it's derived from CCmdTarget. This is important because there may be times when you want to be able to supply a user interface for the data source itself. For example, imagine that you're writing an application to collect real-time data from some sort of scientific instrument. By deriving CDocument from CCmdTarget, you can send such commands as "Start Instrument" and "Stop Instrument" to the instrument (a.k.a. the data source or the document).

Another advantage is not only sending programmatic commands, but also allowing the user interface to directly bind to functionality in the document. In other words, you can implement command handlers in the document itself. Because of command routing, the commands find their way to the document from user-interface elements, such as toolbars and menus, which actually belong to totally different objects. This fact can be abused to make dangerous access to the document, but if used correctly, it can be a very valuable technique.

Views

An application with only a document would be fairly uninteresting. There has to be some way of rendering the data on the screen as well as providing a user interface for manipulating the data. That functionality is supplied by MFC's CView class.

In an MFC program using the document/view architecture, views are responsible for rendering a document's data. CView is derived from CWnd, which is derived from CCmdTarget, which is derived from CObject. (Sounds kind of like the Bible, doesn't it.). Because CView is derived from CCmdTarget, it's eligible to receive command messages from menus and controls (a good thing, since the view is often responsible for handling the user interface as well). Because CView is derived from CWnd, it's also eligible for receiving window messages (like WM_PAINT and WM_SIZE).

In a standard C/SDK application, you render the data by handling the WM_PAINT message. You create a handle to a device context (an HDC) by calling BeginPaint().

Once you have a device context, there are many functions available for drawing in the window.

Standard MFC applications using the document/view architecture do their drawing within a view. CView includes a function called `OnDraw()`, which is called by the framework whenever a view needs to be updated.

MFC views are simply borderless windows that supply data renderings. However, when you use an MFC-based application, the drawings appear within a window that has a border and menus. That window is called a frame window, and it's necessary for the document/view architecture to work properly.

For all practical purposes, a CView is the same as a child CWnd with some extra virtual functions. So, if your CView derivative is written correctly, it can be used in a somewhat abnormal situation (for example, as a child window in a dialog). In general there is nothing in CView that depends on its parent being a CFrameWnd, although some features are disabled when the parent is not a frame.

Document/View Frames

So far, we have described the way the document/view framework isolates data management and rendering in two places—the document and the view. However, there's one other important aspect to this scenario: the ability to apply a different user interface (that is, a different set of menus and controls) to each separate view.

Imagine writing a spreadsheet application that has a single document (the spreadsheet itself) but renders the data in two different ways. Perhaps this application displays the data in a grid (the first view) and as a graph (in the second view). Under this scenario, you may want to have different menus for each view. Perhaps you want to include such menu options as sorting and querying data for the grid view. For the graph view, you may want to include menu options like changing graph colors and formats. Fortunately, each view's frame handles this user-interface code.

At first glance, you may wonder why the menu management is handled by a separate frame and not by the view itself. It's good design practice to isolate this type of functionality and not tightly couple it. That way, you reduce dependencies, which makes it much easier to work on a single piece of code at a time. This technique is also the cornerstone for isolating the differences between SDI, MDI, and OLE in-place editing. Separating the CView and the frame makes CViews more flexible, so that you can use them in other situations.

Figure 7-1 illustrates the relationships between frames and views.

Single Document Interface (SDI) and Multiple Document Interface (MDI) applications handle their documents, views, and frames slightly differently. Of course, SDI applications concern themselves with only one document at a time. Good examples of this kind of application are Microsoft Paint and Microsoft WordPad, applets that come with Windows. You can view only one document at a time. Whenever you

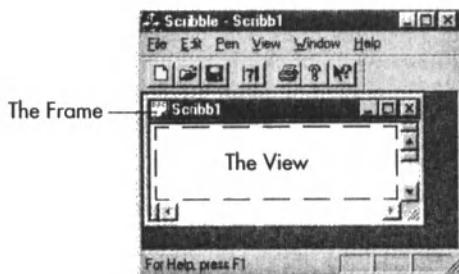


Figure 7-1. A view within a frame window

open a new document while one is already open, the first document is closed before the second one is opened. Applications using MDI can open many documents at the same time. A good example of an MDI application is Microsoft Excel or Novell's Quattro Pro.

SDI applications use a class derived from CFrameWnd as the frame housing the view. MDI applications derive a class from CMDIChildWnd as the frame housing the document's views.

Document Templates

One interesting aspect of the document/view architecture is that you work with the three previous components as a unit. In other words, the ideas of a document, its renderings, and its user interface are treated together as a whole. Document templates tie the whole picture together. These three components are managed by a class called CDCTemplate.

Document Templates: Do You Have to Use Them?

You can use frames/views/documents without relying on CDCTemplate or any of its derivatives. CDCTemplate really doesn't do anything special other than encapsulate a few common uses of the document/view framework. Special applications most often require special document template classes or something totally different. Also, the MFC stock implementation of many commands, such as File|New and File|Open, relies on the document template list. There is nothing stopping you from implementing your own version of these commands.

Your application has one document template for each type of document that it supports. For example, if your application supports both a spreadsheet and a database, the application has two document template objects. Each document template is responsible for creating and managing all the documents of its type.

CDCTemplate is an abstract base class that defines the base functionality for handling documents, frames, and views. GetFirstDocPosition(), GetNextDocPosition(),

and OpenDocumentFile() are the pure virtual functions that make CDocTemplate an abstract base class. MFC defines two other document template classes that can be used directly within applications. They are CSingleDocTemplate and CMultiDocTemplate.

CSingleDocTemplate

For applications that use only one kind of document, MFC defines the CSingleDocTemplate class. The CSingleDocTemplate's constructor takes four arguments: (1) a resource ID identifying various resources that are tied to the document template, (2) the run-time class of the CDocument-derived class, (3) the run-time class of the view's frame, and (4) the run-time class of the document's view.

The resource ID also identifies an entry into the application's string table. It specifies a single string that includes seven specific pieces of information about the document: (1) the window title, (2) the document name, (3) a description of the type of document for use when creating a new document, (4) a description of the file type for the standard File Open dialog, (5) a file extension filter, (6) a description of file type for use by the File Manager, and (7) the ProgID that is registered in the Windows registry.

CMultiDocTemplate

MFC defines a class called CMultiDocTemplate for applications using more than one kind of document. CMultiDocTemplate is very similar to CSingleDocTemplate. Its constructor takes the same arguments. The main difference with the CMultiDocTemplate is that it can hold a list of documents, whereas the CSingleDocTemplate holds only one document.

CWinApp's Role

MFC defines two classes to manage document/view/frame combinations: CSingleDocTemplate and CMultiDocTemplate. But who manages the document templates? In an MFC application, the CWinApp object manages the document templates.

CWinApp::InitInstance() is the usual place for creating document templates. AppWizard will start you off with a single document template that ties together a document, a frame, and a view. Right after the document template is constructed, InitInstance() adds the document template to CWinApp's list of document templates, using CWinApp::AddDocTemplate().

How does CWinApp use document templates? In a standard MFC application, the application object (derived from CWinApp) handles the File|New and the File|Open commands. You might think the framework creates the document directly. In fact, what happens is that the document templates handle creating new files and opening existing files.

CWinApp has functions for responding to menu items for handling files. They are `OnFileNew()` and `OnFileOpen()`. You'll find these functions in the CWinApp class. To get them working, just add water—just add message map entries. Of course, if you create your application using AppWizard, AppWizard adds the message map entries for you. Once there are message map entries for `OnFileNew()` and `OnFileOpen()`, CWinApp handles the rest. These implementations might not be appropriate for your application. If so, remember that they are just normal message handlers that can be overridden or replaced just like any other command handler.

Now that we've had a whirlwind overview of MFC's document/view architecture, let's see what makes it tick.

Inside the Document/View Architecture

In this section we dive into the document/view architecture classes (using CWinApp as our springboard) and reveal how they all fit together. Then we cover some advanced document/view issues, such as printing and print preview and CView-derivative internals. There are a lot of interdependencies among the many document/view architecture classes. Understanding the details will enable you to use document/view effectively.

CWinApp manages document templates, and document templates manage frames/views/documents. The first step in understanding the document/view hierarchy is learning the interface between CWinApp and CDocTemplate.

The CWinApp/CDocTemplate Interface: CDocManager

Because MFC applications can register more than one type of document template, the framework has to maintain a list of the document templates. That raises an important question: Where does this list live? It seems logical for the list of document templates reside in CWinApp, but the CWinApp declaration, in AFXWIN.H, has no lists of document templates.

However, in that declaration, there is a data member that reveals the link between CWinApp and CDocTemplates. The relevant section from the header is

```
CDocManager * m_pDocManager;
```

CDocManager is currently (in MFC 4.0) an undocumented class, so let's look at its declaration to see what it is doing. Listing 7-1 contains the declaration for CDocManager.

Listing 7-1. The undocumented CDocManager declaration, from AFXWIN.H

```

class CDocManager : public CObject
{
    DECLARE_DYNAMIC(CDocManager) public:
// Constructor
    CDocManager();
//Document functions
    virtual void AddDocTemplate(CDocTemplate* pTemplate);
    virtual POSITION GetFirstDocTemplatePosition() const;
    virtual CDocTemplate* GetNextDocTemplate(POSITION& pos) const;
    virtual void RegisterShellFileTypes(BOOL bWin95);
    virtual CDocument* OpenDocumentFile(LPCTSTR
                                         lpszFileName); // open named file
    virtual BOOL SaveAllModified(); // save before exit
    virtual void CloseAllDocuments(BOOL bEndSession); // close documents
before exit
    virtual int GetOpenDocumentCount();
// helper for standard commdlg dialogs
    virtual BOOL DoPromptFileName(CString& fileName, UINT nIDSTitle,
                                  DWORD lFlags, BOOL bOpenFileDialog,
                                  CDocTemplate* pTemplate);
//Commands ** Some omitted.
    virtual void OnFileNew();
    virtual void OnFileOpen();
// Implementation
protected:
    CPtrList m_templateList;
public:
    virtual ~CDocManager();
};

```

In CDocManager, if you skip right to the // Implementation section, you will notice a CPtrList (a list of CObject pointers) called m_templateList. There's the missing list of document templates! Most of the member functions in CDocManager manage this list and provide CWinApp an interface for CDocTemplates.

Version Note

Versions of MFC before release 4.0 maintained the document template list right in CWinApp. Microsoft appears to be abstracting it out into CDocManager, though it's not crystal clear why. Perhaps they are making way for new MDI alternatives. Who knows—maybe managing this list in CWinApp got cumbersome and Microsoft just wanted to dust off CWinApp in 4.0.

Here's what our MFC insider had to say:

The main reason we separated out this implementation into a separate class is so that applications which don't use doc/view (simple utilities, for example) don't take on the overhead. In

the early days, it was OK to put all this functionality into CWinApp, because it was relatively small, such that it didn't impact the size of applications that didn't use it too much. Over time the implementation has grown as we added new features. In MFC 4.0 we separate out this functionality into a separate class with all virtual functions. If a CDocManager object is never created, its virtual table is never referenced, and as a result, its functions are not linked in. You'll see this technique used in several cases (for example, COccManager and CRecentFileList) where we want a feature and its associated code to be optional. Of course, if you use the DLL then all the code is there and this trick doesn't really do a lot of good. If you statically link, this can make a difference in the size of the resulting application (we keep an eye on the size of the "HelloApp" sample for doing these kind of optimizations). Also, as a side effect, someone could replace the CDocManager implementation with one of their own. In the future, we do want to reduce the amount of stuff in CWinApp—it is getting out of hand—and this is one example of moving toward that goal as well.

Because most of CDocManager's functionality used to live in CWinApp, many of the CWinApp member functions (especially those that deal with document templates) now just call right through the m_pDocManager pointer.

For example, here's the implementation of CWinApp::AddDocTemplate(), from APPUI2.CPP:

```
void CWinApp::AddDocTemplate(CDocTemplate* pTemplate)
{
    if (m_pDocManager == NULL)
        m_pDocManager = new CDocManager;
    m_pDocManager->AddDocTemplate(pTemplate);
}
```

Here's a list of CWinApp member functions that call directly through the CWinApp::m_pDocManager CDocManager pointer.

- CWinApp::AddDocTemplate()
- CWinApp::CloseAllDocuments()
- CWinApp::DoPromptFileName()
- CWinApp::GetFirstDocTemplatePosition()
- CWinApp::GetNextDocTemplate()
- CWinApp::GetOpenDocumentCount()
- CWinApp::OnFileNew()
- CWinApp::OnFileOpen()
- CWinApp::OpenDocumentFile()
- CWinApp::RegisterShellFileTypes()
- CWinApp::SaveAllModified()

Most of these member functions are well documented, so we will skip right to the data members of CDocManager. Don't worry, though: when we look at how all of the document/view components are created, we will visit some of these CDocManager member functions.

- `m_templateList`—As discussed, this pointer maintains the list of document templates. One interesting note is that this data member is declared protected. To get to it, your object would have to be either a friend or a derivative of CDocManager. Unfortunately, this makes it near impossible to get to this pointer “directly” from your MFC applications. You can easily create a CDocManager derivative, but the hard part is replacing the `m_pDocManager` pointer that is in CWinApp to use your CDocManager derivative.

Luckily, the `GetFirstDocTemplatePosition()` and `GetNextDocTemplate()` CWinApp member functions let you iterate through the template list. There aren't any cases (that we can think of) where you would need direct access to `m_pDocManager`, because you can get to it through member functions.

Inside CDocManager::OnFileNew()

One critical CDocManager member function is `OnFileNew()`. `OnFileNew()` is responsible for creating a new document template (and consequently all of the other objects involved in document/view). Let's go over the pseudocode for `CDocManager::OnFileNew()`, which is in Listing 7-2.

Listing 7-2. The CDocManager::OnFileNew() pseudocode, from DOCMGR.CPP

```
void CDocManager::OnFileNew()
{
    CDocTemplate* pTemplate = (CDocTemplate*)m_templateList.GetHead();
    if (m_templateList.GetCount() > 1) {
        CNewTypeDlg dlg(&m_templateList);
        int nID = dlg.DoModal();
        if (nID == IDOK)
            pTemplate = dlg.m_pSelectedTemplate;
        else
            return;      // none - cancel operation
    }
    pTemplate->OpenDocumentFile(NULL);
}
```

First, `OnFileNew()` checks to see if there is more than one document template in the `m_templateList`. If so, `OnFileNew()` creates a `CNewTypeDlg` dialog, which presents the user with a list of documents to choose from.

Once the user chooses the appropriate document template, OnFileNew() calls CDocTemplate::OpenDocumentFile(). We'll look at CDocTemplate::OpenDocumentFile() when we look at CDocTemplate.

Notice that in OnFileNew(), CNewTypeDlg takes a pointer to the document template pointer list, m_templateList, in its constructor. Also note that after DoModal(), OnFileNew() gets the selected dialog template from CNewTypeDlg::m_pSelectedTemplate.

CNewTypeDlg is an interesting example of a quick and easy way to present the end user with a list of choices (in this case, document types). Let's take a short side trip and look at how this dialog is implemented.

A CDocManager Aside: The Undocumented Helper Class CNewTypeDlg

Figure 7-2 shows CNewTypeDlg in action. The declaration for CNewTypeDlg lives in DOCMGR.CPP and is reproduced in Listing 7-3.

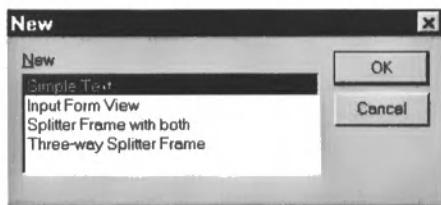


Figure 7-2. The CNewTypeDlg

Listing 7-3. The CNewTypeDlg declaration, from DOCMGR.CPP

```
class CNewTypeDlg : public CDialog
{
protected:
    CPtrList*    m_pList;
public:
    CDocTemplate*    m_pSelectedTemplate;
    enum { IDD = AFX_IDD_NEWTYPEDLG };
    CNewTypeDlg(CPtrList* pList) : CDialog(CNewTypeDlg::IDD){
        m_pList = pList;
        m_pSelectedTemplate = NULL;
    }
    ~CNewTypeDlg() { }
protected:
    BOOL OnInitDialog();
    void OnOK();
    DECLARE_MESSAGE_MAP()
};
```

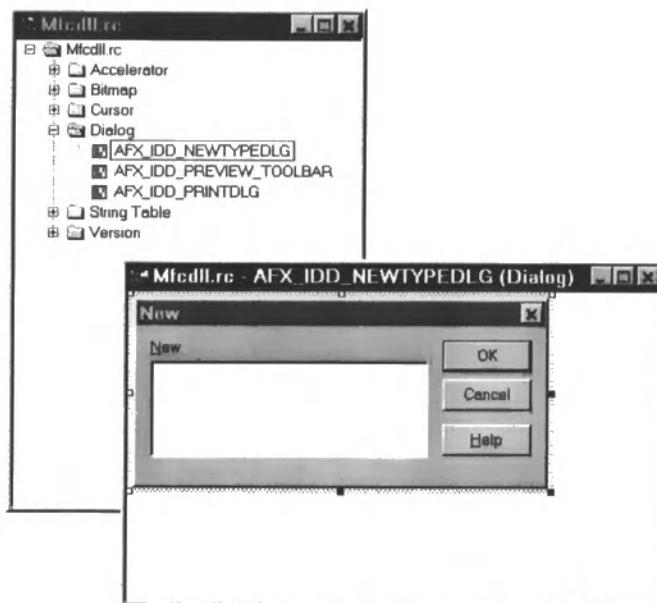


Figure 7-3. The new dialog resource

From the CNewTypeDlg declaration, we can tell that CNewTypeDlg is a CDIALOG derivative and that it has two data members, *m_pList* and *m_pSelectedTemplate*. The dialog has resource ID AFX_IDD_NEWTYPEDLG. This dialog resource is defined in INCLUDE\AFXRES.RC, and its layout is pictured in Figure 7-3.

Looking further at the inline constructor, we see that *m_pList* points to the list of document templates passed into the constructor.

Now let's see how CNewTypeDlg goes about displaying the list of document templates and returning the selection. The pseudocode for CNewTypeDlg::OnInitDialog() is in Listing 7-4.

Listing 7-4. The CNewTypeDlg::OnInitDialog() pseudocode, from DOCMGR.CPP

```
BOOL CNewTypeDlg::OnInitDialog()
{
    CLISTBOX* pListBox = (CLISTBOX*)GetDlgItem(AFX_IDC_LISTBOX);
    POSITION pos = m_pList->GetHeadPosition();
    while (pos != NULL) {
        CDocTemplate* pTemplate = (CDocTemplate*)m_pList->GetNext(pos);
        CString strTypeName;
        if (pTemplate->GetDocString(strTypeName,
            CDocTemplate::fileNewName) &&
            !strTypeName.IsEmpty()) {
            // add it to the listbox
            int nIndex = pListBox->AddString(strTypeName);
```

```

    pListBox->SetItemDataPtr(nIndex, pTemplate);
} // end if
} // end while
return CDialog::OnInitDialog();
}

```

First, OnInitDialog() gets a pointer to the CListBox by calling GetDlgItem() and typecasting it to a CListBox pointer.

After obtaining the CListBox pointer, OnInitDialog() iterates through its list of document templates. While iterating, OnInitDialog() calls CDocTemplate::GetDocString() to get a string version of the document type. OnInitDialog() takes the string for each document, adds it to the list, and then calls CListBox::SetItemDataPtr() to attach the document template pointer to the list item.

Once OnInitDialog() is done adding the document templates to the list, it returns CDialog::OnInitDialog().

In CNewType::OnOK(), when the user selects an item from the list and presses the OK button, CNewType::OnOK() gets the selected item from the list and then gets a pointer to the document template by calling CListBox::GetItemDataPtr(). OnOK() stores the selected template pointer in m_pSelectedTemplate so that it can be retrieved after DoModal().

Now if you ever find yourself in a similar situation, you will know a quick and easy way to display a list of choices in a modal dialog.

At this point, we've exhausted the interesting CDocManager internals, so let's move down the document/view chain of command to the document template.

CDocTemplate: CDocument, CView, and CFrameWnd Manager

As previously stated, CDocTemplate has the job of tying together a document/view/frame trio. The declaration for CDocTemplate can be found in AFXWIN.H, and the implementation is in DOCTEMPL.CPP. Remember, CDocTemplate is an abstract base class and your MFC applications actually use one of the two CDocTemplate derivatives, CSingleDocTemplate or CMultiDocTemplate. CSingleDocTemplate lives in DOCSINGL.CPP, and CMultiDocTemplate lives in DOCMULTI.CPP.

Let's see how CDocTemplate does its job by taking a peek at its declaration, from AFXWIN.H as shown in Listing 7-5. Note: CDocTemplate has an extremely long declaration, so we've trimmed out many OLE and other members for brevity.

Listing 7-5. A very abbreviated CDocTemplate declaration

```

class CDocTemplate : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocTemplate)

```

```

// Constructors ** omitted.
// Attributes ** omitted
// Operations ** omitted
// Overridables ** omitted
// Implementation ** some omitted

protected:
    UINT m_nIDResource;           // IDR_ for frame/menu/accel as well
    CRuntimeClass* m_pDocClass;    // class for creating new documents
    CRuntimeClass* m_pFrameClass;  // class for creating new frames
    CRuntimeClass* m_pViewClass;   // class for creating new views
    CString m_strDocStrings;      // '\n' separated names
};

}

```

The abbreviated CDocTemplate declaration highlights the undocumented CDocTemplate data members. A brief description of each data member and its purpose follows:

- **m_nIDResource**—Stores the first argument to the CDocTemplate constructor, which is the resource ID that references the various resources tied to the document template.
- **m_pDocClass**—A pointer to the CRuntimeClass structure of the CDocument (or derivative) class for this document template. Set in the constructor.
- **m_pFrameClass**—A pointer to the CRuntimeClass structure of the CFrameWnd (or derivative) class for this document template. Set in the constructor.
- **m_pViewClass**—A pointer to the CRuntimeClass structure of the CView (or derivative) class for this document template. Set in the constructor.
- **m_strDocStrings**—A CString that contains the string table resource that defines the seven strings associated with every document template (as covered earlier). Initialized in LoadTemplate().

As we look at key CDocTemplate member functions, we will see more details about how these data members are used.

How Does CDocTemplate Create New Documents/Views/Frames?

The answer to this very important question lies in two member functions: CDocTemplate::CreateNewDocument() and CDocTemplate::CreateNewFrame().

As the names imply, CreateNewDocument() creates a document, and CreateNewFrame() creates a frame. But that leaves the view. Where does the view get created? We'll leave view creation as a mystery for now.

Let's look first at CreateNewDocument() and then at CreateNewFrame(). Listing 7-6 shows the pseudocode for CreateNewDocument().

Listing 7-6. The pseudocode for CDocTemplate::CreateNewDocument(), from DOCTEMPL.CPP

```
CDocument* CDocTemplate::CreateNewDocument()
{
    CDocument* pDocument = (CDocument*)m_pDocClass->CreateObject();
    if (pDocument == NULL) {
        TRACE1("Warning: Dynamic create of document type %s failed.\n",
               m_pDocClass->m_lpszClassName);
        return NULL;
    }
    AddDocument(pDocument);
    return pDocument;
}
```

First of all, CreateNewDocument() creates a new document by calling CRuntimeClass::CreateObject() via the CRuntimeClass pointer data member m_pDocClass. If you don't remember the details of how MFC dynamic creation works, you may want to look back at Chapter 5. The MFC document/view architecture uses dynamic creation heavily.

After storing the new CDocument (or derivative) pointer in pDocument, CreateNewDocument() verifies that the CreateObject() worked. If the CreateObject() call does not work, CreateNewDocument() generates some debug output and then returns NULL, indicating failure. If the CreateObject() call does work, CreateNewDocument() adds the document to the list of open documents by calling AddDocument(). Finally, CreateNewDocument() returns the pointer to the new document.

Hmmm, well, it's pretty clear that CreateNewDocument() is not creating the view. Let's keep on the trail and look at CreateNewFrame() for hints. Listing 7-7 contains the pseudocode for CreateNewFrame().

Listing 7-7. The CDocTemplate::CreateNewFrame() pseudocode, from DOCTEMPL.CPP

```
CFrameWnd* CDocTemplate::CreateNewFrame(CDocument* pDoc, CFrameWnd* pOther)
{
    CCreateContext context;
    context.m_pCurrentFrame = pOther;
    context.m_pCurrentDoc = pDoc;
    context.m_pNewViewClass = m_pViewClass;
    context.m_pNewDocTemplate = this;
    CFrameWnd* pFrame = (CFrameWnd*)m_pFrameClass->CreateObject();
    // create new frame from resource
    if (!pFrame->LoadFrame(m_nIDResource,
```

```

WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, // default frame styles
NULL, &context));
    TRACE0("Warning: CDocTemplate couldn't create a frame.\n");
    return NULL;
}
return pFrame;
}

```

First, CreateNewFrame() fills out a CCreateContext (see the next section for details) with the information relevant to the frame being created. pOther is used only in the case of MDI and is not important to this discussion.

Next, CreateNewFrame() creates a frame from the CRuntimeClass structure in m_pFrameClass using CRuntimeClass::CreateObject(). After creating the frame, CreateNewFrame() calls LoadFrame() to load the frame's resources and create the frame based on those values. Finally, CreateNewFrame() returns a pointer to the new frame.

Before continuing our quest to find where the view is created, let's take a side trip and look at CCreateContext.

CCreateContext: The Creation Helper

CCreateContext is a helper structure used by the framework to store information critical for creating various elements of the document/view architecture. Listing 7-8 contains the CCreateContext declaration.

Listing 7-8. The CCreateContext declaration, from AFXEXT.H

```

struct CCreateContext // Creation information helper
{
    CRuntimeClass* m_pNewViewClass;
    CDocument* m_pCurrentDoc;
    CDocTemplate* m_pNewDocTemplate;
    CView* m_pLastView;
    CFrameWnd* m_pCurrentFrame;
// Implementation
    CCreateContext();
};

```

Here's a summary of what each field of CCreateContext stores:

- m_pNewViewClass—A CRuntimeClass pointer used for creating views.
(Hmmm . . .)
- m_pCurrentDoc—Points to the current document object.
- m_pCurrentFrame—Points to the current frame object.
- m_pNewDocTemplate—if there are multiple documents, points to the last one.
- m_pLastView—if there are multiple views, points to the last view.

This is certainly some useful information, but it all lives in the document template already, so why bother stuffing everything into this CCreateContext structure? The problem is that MFC needs to pass around this information among the document template, the frame, the view, and the document during creation. Instead of making this information global, MFC stores it in the structure and passes it around using a pointer to the instance we saw being created in CDockTemplate::CreateNewFrame(). You will see CCreateContext pop up over and over again throughout the various document/view topics covered. MFC sometimes needs to pass a pointer to a CCreateContext structure through the LPARAM of a message, so don't be alarmed if you see something like this:

```
int CView::OnCreate(LPCREATESTRUCT lpcs)
{
    //...
    CCreateContext* pContext = (CCreateContext*)lpcs->lpCreateParams;
    //...
}
```

CView Creation?

CCreateContext::m_pNewViewClass gives us our first solid clue about view creation. This clue will eventually take us to the CFrameWnd class, but for now there are still some CDockTemplate details you need to know about. We will have to pick up the CView creation trail later in the CFrameWnd internals coverage.

CSingleDocTemplate and CMultiDocTemplate

After our discussion of CreateNewDocument() and CreateNewFrame(), you might be asking Where does this list of open documents live? It definitely was not in the CDockTemplate declaration. The list of documents is different for each of the two CDockTemplate derivatives, CSingleDocTemplate and CMultiDocTemplate. Let's look at those two derivatives and see how they manage the list of open documents.

In the declaration for CSingleDocTemplate, you will see that it has a data member CDocument pointer declared as this:

```
CDocument* m_pOnlyDoc;
```

(Kind of a lonely sounding data member, isn't it?)

On the other hand, CMultiDocTemplate has to maintain more than one open document, so its declaration has this:

```
CPtrList m_docList;           // open documents of this type
```

which is a list of CObject derivatives, or more precisely, open CDocuments.

In the case of CSingleDocTemplate, AddDocument() sets m_pOnlyDoc to the argument. CMultiDocTemplate::AddDocument() adds the new document to its list by calling m_docList.AddTail(pDocument).

In addition to m_docList, CMultiDocTemplate has one other data member:

```
int m_nUntitledCount;
```

This data member is used to keep track of the number of untitled windows. When the user creates several unnamed documents, MFC automatically calls them Untitled[X], where X indicates the number of the unnamed document.

CMultiDocTemplate::OpenDocumentFile()

See CMultiDocTemplate::OpenDocumentFile() to understand how a CDocTemplate opens a document. CMultiDocTemplate has a couple more interesting caveats than CSingleDocTemplate, so we decided to look at CMultiDocTemplate::OpenDocumentFile() instead. Listing 7-9 contains the OpenDocumentFile() pseudocode, from DOCMULTI.CPP.

Listing 7-9. The CMultiDocTemplate::OpenDocumentFile() pseudocode, from DOCMULTI.CPP.

```
CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
BOOL bMakeVisible)
{
    // Hey reader-> Remember these from earlier?!
    CDocument* pDocument = CreateNewDocument();
    CFrameWnd* pFrame     = CreateNewFrame(pDocument, NULL);
    if (lpszPathName == NULL) { // create a new document - with default
                                // document name
        SetDefaultTitle(pDocument);
        if (!pDocument->OnNewDocument()) {
            pFrame->DestroyWindow();
            return NULL;
        }
        m_nUntitledCount++;
    }
    else {      // open an existing document
        CWaitCursor wait;
        if (!pDocument->OnOpenDocument(lpszPathName)){
            pFrame->DestroyWindow();
            return NULL;
        }
        pDocument->SetPathName(lpszPathName);
    }
    InitialUpdateFrame(pFrame, pDocument, bMakeVisible); return pDocument;
}
```

`OpenDocumentFile()` calls `CreateNewDocument()` to create a fresh, new empty document (a `CDocument` derivative). Next, a new frame is created by calling `CreateNewFrame()`. The `lpszPathName` argument determines what happens next.

If `lpszPathName` is `NULL`, `OpenDocumentFile()` knows to create an empty “new” document. It does this by calling `SetDefaultTitle()`, which determines a name for the new document (using a combination of `Untitled`, or a designated name, plus the untitled count from `m_nUntitledCount`) and then sets the name in the document by calling `CDocument::SetTitle()` via the new document pointer. Finally, `OpenDocumentFile()` calls `OnOpenDocument()` and then increments the untitled count data member.

If `lpszPathName` is not `NULL`, `OpenDocumentFile()` attempts to open the specified file. First, `OpenDocumentFile()` displays a wait cursor and then calls `CDocument::OnOpenDocument()`, passing the file name as an argument. If `CDocument::OnOpenDocument()` fails, `OpenDocumentFile()` destroys the created frame and returns `NULL`, indicating failure. On the other hand, if `CDocument::OnOpenDocument()` doesn’t fail, `OpenDocumentFile()` calls `CDocument::SetPathName()` to update the path of the `CDocument`.

Regardless of the state of `lpszPathName`, `OpenDocumentFile()` concludes by calling `CDocTemplate::InitialUpdateFrame()`, which calls through to `CFrameWnd::InitialUpdateFrame()`, passing along a pointer to the document.

Though we’ve already covered a good deal of `CFrameWnd` in Chapter 2, let’s look at the areas of that class that are specific to documents/views so we can see how it fits into the bigger picture.

CFrameWnd Internals

There are several instances in which `CFrameWnd` comes in contact with the `CDocument` and `CView` classes (at least from the `CFrameWnd` side of things).

1. `CView` creation!! Yeah! (The wait is over . . .) We won’t reveal how this works quite yet.
2. `CFrameWnd` has a `CView` pointer data member, `m_pViewActive`, that it uses to communicate activations and deactivations with the currently active view. This data member can be retrieved via `CFrameWnd::GetActiveView()` and set via `CFrameWnd::SetActiveView()`.
3. The `InitialUpdateFrame()` member function causes the `CView::OnInitialUpdate()` member function of all active views to be called. (Splitter windows covered in Chapter 9 allow for multiple views to live in a single frame.)

CFrameWnd and CView Creation

And now the moment you’ve all been waiting for: when and how `CViews` are created. When we last left this mystery, we had just discovered that

`CDocTemplate::CreateNewDocument()` creates the document and `CDocTemplate::CreateNewFrame()` creates the frame window. Also, remember that `CreateNewFrame()` squirreled away the view CRuntimeClass information in `CCreateContext::m_pNewViewClass` and then passed the structure to `CFrameWnd::LoadFrame()`.

In `WINFRM.CPP` there is only one member function that uses `CCreateContext::m_pNewViewClass`: that's `CFrameWnd::CreateView()`.

Let's look at it next.

View Creation: `CFrameWnd::CreateView()`

Listing 7-10 contains the `CFrameWnd::CreateView()` pseudocode.

Listing 7-10. The `CFrameWnd::CreateView()` pseudocode, from `WINFRM.CPP`

```
CWnd* CFrameWnd::CreateView(CCreateContext* pContext, UINT nID)
{
    CWnd* pView = (CWnd*)pContext->m_pNewViewClass->CreateObject();
    if (pView == NULL)
        return NULL;
    if (!pView->Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0,0,0,0), this, nID, pContext))
        return NULL;           // can't continue without a view
    return pView;
}
```

Right off the bat, `CreateView()` creates a view using the CRuntimeClass stored in `CCreateContext::m_pNewViewClass`. After creating the view, it calls `CWnd::Create()` to perform the second-phase creation of the MFC object.

After all that searching, the actual creation is pretty simple. But . . .

Wait, There's More!

Now the big question is, Who calls `CreateView()`? Remember that `CreateNewDocument()` and `CreateNewFrame()` were called by `C[Single/Multi]DocTemplate::OpenDocumentFile()`, but `OpenDocumentFile()` never calls `CreateView()`.

To find the answer, let's trace our way up the calling stack. In a search through `WINFRM.CPP`, the only function that calls `CreateView()` is `OnCreateClient()`. `OnCreateClient()` is called only by `OnCreateHelper()`. `OnCreateHelper()` is called only by `OnCreate()`.

`CFrameWnd::OnCreate()` is a message map entry that is called when the frame receives a WM_CREATE message. So when the document template creates the frame in `CreateNewFrame()`, Windows sends a WM_CREATE, which calls

```
OnCreate() -> OnCreateHelper() -> OnCreateClient() -> CreateView().
```

(Actually, splitter windows call CreateView() too, but you will have to wait until Chapter 9 to see how and why.)

Now that you know how views are created, it's time to continue our coverage of CFrameWnd and how it fits into documents/views. We already explained the role of `m_pViewActive` in the section introduction, so that leaves CFrameWnd::InitialUpdateFrame().

CFrameWnd::InitialUpdateFrame()

Recall that InitialUpdateFrame() causes all views in the frame to be updated. Let's see how it accomplishes that. Listing 7-11 contains the pseudocode for CFrameWnd::InitialUpdateFrame().

Listing 7-11. The CFrameWnd::InitialUpdateFrame() pseudocode, from WINFRM.CPP

```
void CFrameWnd::InitialUpdateFrame(CDocument* pDoc, BOOL bMakeVisible)
{
    CView* pView = // ** omitted it determines the active view or gets
    // first splitter window - See Chapter 7.
    if (bMakeVisible) {
        SendMessageToDescendants(WM_INITIALUPDATE, 0, 0, TRUE, TRUE);
        ActivateFrame();
        pView->OnActivateView(TRUE, pView, pView);
    }
    // update frame counts and frame title (may already have been visible)
    if (pDoc != NULL)
        pDoc->UpdateFrameCounts();
    OnUpdateFrameTitle(TRUE);
}
```

First, InitialUpdateFrame() gets a pointer to the active view and stores it in `pView`. Next, if the `bMakeVisible` argument is TRUE, InitialUpdateFrame() sends the MFC-specific message, WM_INITIALUPDATE, to all of the descendants (this includes all views). For now, you just need to know that it will result in the WM_INITIALUPDATE message getting sent to all of the views in the frame window. MFC maps this message to CView::OnInitialUpdate(). After sending the WM_INITIALUPDATE message, InitialUpdateFrame() activates the frame and the view. It activates the view by calling CView::OnActivateView().

Next, InitialUpdateFrame() updates a frame count in the document (more on this coming up), and finally, InitialUpdateFrame() updates the title of the frame to reflect the name of the document.

Now the only variables in the equation are CDocument and CView. Let's start with CDocument and then finish up with CView internals.

CDocument Internals

CDocument is rich with undocumented data members, member functions, and just plain interesting behavior. Let's first look at the CDocument declaration, from AFXWIN.H, as shown in Listing 7-12. The source code for CDocument is in DOCCORE.CPP.

Listing 7-12. The abbreviated CDocument declaration, from AFXWIN.H

```
class CDocument : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocument)
// Constructors **omitted
// Attributes **omitted
// Operations **omitted
// Overridables **omitted
// Implementation **Some ommitted
protected:
    CString m_strTitle;
    CString m_strPathName;
    CDocTemplate* m_pDocTemplate;
    CPtrList m_viewList;           // list of views
    BOOL m_bModified;            // changed since last saved
public:
    BOOL m_bAutoDelete;          // TRUE => delete document when no more views
    virtual ~CDocument();
// implementation helpers
    virtual BOOL DoSave(LPCTSTR lpszPathName, BOOL bReplace = TRUE);
    virtual BOOL DoFileSave();
    virtual void UpdateFrameCounts();
    void DisconnectViews();
    void SendInitialUpdate();
    friend class CDocTemplate;
// Message handlers **omitted
    DECLARE_MESSAGE_MAP()
};
```

Wow! There's an unprecedented number of undocumented members in CDocument. We won't be able to focus on them all, so here is a brief summary of what each does.

- **m_strTitle**—The name of the document. Set with SetTitle(). Used to specify the title of the frame window as well as the name of the file when saved.
- **m_strPathName**—The path of the current document (including the file name). Set with SetPathName(). This member is used for additions to the MRU menu, and is used when saving the file.
- **m_pDocTemplate**—A pointer to the document's document template. (See the sidebar for more information.) Set in CDocTemplate::AddDocument(). You can access this member through CDocument::GetDocTemplate().

- **m_viewList**—Every MFC document can have an unlimited number of views. m_viewList keeps pointers to all of the views open on the document. CDocument uses m_viewList to let views know when the document has changed and thus cause them to update to reflect the document changes. Views are added to m_viewList with AddView() and RemoveView(). You can iterate the list of views stored in m_viewList using GetFirstViewPosition() and GetNextView(). Documents also use m_viewList as a way to access frame windows. See the description of UpdateFrameCounts() that follows.
- **m_bModified**—A “dirty” flag that when set indicates that the document has been changed by the user. When this flag is set, the framework will prompt the user before closing the document. You can check the state with IsModified() and set the modified flag via SetModifiedFlag().
- **m_bAutoDelete**—This flag is set to TRUE by the framework, indicating that a document object should be deleted when all views on it are closed. The developer can set this to FALSE in order to keep the document object around even if there aren’t any open views on the document.
- **DisconnectViews()**—This implementation member function iterates through the view list stored in m_viewList and sets the CView::m_pDocument pointer to NULL, thus disconnecting all views from the document. DisconnectViews() is called in the CDocument destructor.
- **DoSave()**—This member function contains the “core” code for saving a document. CDocument members OnFileSaveAs() and OnFileSave() both eventually call DoSave() after prompting for a file or verifying file names. After DoSave() prompts for a file, it calls OnSaveDocument().
- **DoFileSave()**—CDocument::OnFileSave() calls this member function. DoFileSave() calls DoSave() after checking some attributes of the file that the document is to be saved in.
- **UpdateFrameCounts()**—This extremely interesting CDocument member is too long to cover in detail here. We highly recommend that you take a second to look at its source in DOCCORE.CPP. UpdateFrameCounts() counts the frames open for all views of the document and also tells them to update their window titles based on the new count. We say UpdateFrameCounts() is so interesting because it illustrates how documents iterate through both views and frames. UpdateFrameCounts() also shows an interesting algorithm called mark and sweep, where it performs two passes through the frame windows. In the first pass it marks the frames and in the second pass it updates the marked frames based on the count from the first pass.
- **SendInitialUpdate()**—Iterates through the list of views and calls CView::OnInitialUpdate().

"Hey, I'm Dead!"

One interesting point about the document/view classes is how they keep tabs on each other. For example, if the user closes a document, how does a document template know that this document is now deceased? The solution employed frequently by the MFC document/view architecture is to keep a pointer to a class that needs to be notified of the document's death (an obituary of sorts). In our example, CDocument keeps a pointer to a CDockTemplate (in the m_pDocTemplate data member). This pointer is set by the "creator," and it is the job of the created object to inform the creator of its demise. In the CDocument/CDocTemplate example, CDockTemplate sets CDocument::m_pDocTemplate in its AddDocument() member function. In the CDocument destructor, CDocument checks the m_pDocTemplate and, if it is non-NULL, calls m_pDocTemplate->RemoveDocument(this) to inform the CDockTemplate that it is being destroyed.

Documents and views use the same technique. Every CView keeps a pointer to its associated CDocument and uses the pointer to alert the CDocument that the CView is being destroyed.

Now that we've introduced you to most of the previously undocumented implementation details of CDocument, let's look into three key aspects of CDocument:

1. Creating documents (both new and from files).
2. Sorting documents.
3. Communicating with views.

Creating Documents

Documents are created either "blank" or from a file. If an empty document is being created, CDocument::OnNewDocument() is called. This member function first calls DeleteContents() to clean out the document. The default implementation of DeleteContents() does nothing; if you want to perform special "cleaning" for new document creation, override DeleteContents() in your CDocument derivative. After calling DeleteContents(), OnNewDocument() sets the modified flag to FALSE and makes sure that m_strPathName is empty.

CDocument::OnOpenDocument() handles creating a document from a file. From the CDockTemplate internals section, recall that C[Multi/Single]DocTemplate::OpenDocumentFile() calls CDocument::OnOpenDocument(). OnOpenDocument() is a key member function, so let's look at the pseudocode and see what it is doing. Listing 7-13 contains the CDocument::OnOpenDocument() pseudocode.

Listing 7-13. The CDocument::OnOpenDocument() pseudocode, from DOCCORE.CPP

```
BOOL CDocument::OnOpenDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
```

```

CFile* pFile = GetFile(lpszPathName,
    CFile::modeRead|CFile::shareDenyWrite, &fe);
// **omitted, checks on pFile, assume it worked..
DeleteContents();
SetModifiedFlag(); // dirty during de-serialize
CArchive loadArchive(pFile, CArchive::load | CArchive::bNoFlushonDelete);
loadArchive.m_pDocument = this;
loadArchive.m_bForceFlat = FALSE;
TRY {
    CWaitCursor wait;
    if (pFile->GetLength() != 0)
        Serialize(loadArchive); // load me
    loadArchive.Close();
    ReleaseFile(pFile, FALSE);
}
CATCH_ALL(e){
    //**omitted, catch for archive failure, see code if interested.
}
END_CATCH_ALL
SetModifiedFlag(FALSE); // start off with unmodified
return TRUE;
}

```

If you read Chapter 5, this should seem pretty familiar to you by now. (Remember Figure 5-1?) `CDocument::OnOpenDocument()` is where the `CArchive` is created and serialization begins.

First, `OnOpenDocument()` opens a file by calling `CDocument::GetFile()`, which just calls `CFile::Open()` with the file name. If the file is successfully opened, `OnOpenDocument()` calls `DeleteContents()` to prepare the document for de-serialization (reading in a new document from persistent store). `OnOpenDocument()` calls `SetModified()` because after the `DeleteContents()` call, the document is “dirty.” `OnOpenDocument()` then sets the document as modified. If serialization fails, this will cause the document to be marked as modified.

Next, `OnOpenDocument()` creates a `CArchive` from the `pFile` and passes the `CArchive` to `CDocument::Serialize()`. (See Chapter 5 for more details on what happens in `CDocument::Serialize()`.) Once the document is de-serialized, `OnOpenDocument()` closes the archive and releases the file.

Finally, `OnOpenDocument()` turns off the modified flag and returns `TRUE`, indicating that the document was successfully loaded from file.

Now let’s look at the other side of the equation and examine how documents are saved to file.

Saving Documents

There are two code paths that cause a document to be saved:

1. `OnFileSave()`—Called in response to the File|Save menu selection.
`DoFileSave()`—Makes sure that the file is not read-only.
`DoSave(file name)`—Calls `OnSaveDocument()` with error handling.
`OnSaveDocument()`—Performs the actual saving to file.
2. `OnFileSaveAs()`—Called in response to the File|Save As menu selection.
`DoSave(NULL)`—NULL argument indicates that `DoSave()` should prompt for a file name. Once the file is specified by the user via a common dialog (created using `CWinApp::DoPromptFileName()`), `OnFileSaveAs()` calls `OnSaveDocument()`.
`OnSaveDocument()`—Performs the actual saving to file.

All Roads Lead to `OnSaveDocument()`

From looking at the two code paths, you can tell that the real meat of saving a document happens inside `OnSaveDocument()`. Let's take a firsthand look at what it's doing. The pseudocode for `CDocument::OnSaveDocument()` is in Listing 7-14.

Listing 7-14. The `CDocument::OnSaveDocument()` pseudocode, from DOCCORE.CPP

```
BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = NULL;
    pFile = GetFile(lpszPathName, CFile::modeCreate |
                    CFile::modeReadWrite | CFile::shareExclusive, &fe);
// **omitted, checks on pFile, assume it's cool.
    CArcive saveArchive(pFile, CArcive::store |
        CArcive::bNoFlushonDelete);
    saveArchive.m_pDocument = this;
    saveArchive.m_bForceFlat = FALSE;
    TRY {
        CWaitCursor wait;
        Serialize(saveArchive);      // save me
        saveArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    CATCH_ALL(e) {
        //**omitted, handling of archive exception.
    }
    END_CATCH_ALL
    SetModifiedFlag(FALSE);      // back to unmodified
    return TRUE;                // success
}
```

It's pretty neat how similarly OnSaveDocument() and OnOpenDocument() are written: they are perfect mirror images of each other!

As in OnOpenDocument(), OnSaveDocument() opens a file by calling GetFile() but uses flags that specify the file is opened for saving and not reading (CFile::modeReadWrite | CFile::modeCreate). Next, OnSaveDocument() creates a CArchive and passes it to the Serialize() member function.

Finally, if everything is serialized correctly, the archive is closed, the modified flag is set to FALSE, and OnOpenDocument() returns TRUE, indicating that the document was successfully saved.

Now let's look at the third interesting CDocument internal: how it communicates with views.

Communicating with Views

As mentioned in the CDocument declaration overview, the m_viewList is the medium through which CDocument communicates with views that are “viewing” the document.

There are really only a couple of situations where a document needs to communicate with its view:

1. Notifying the views that the document has been destroyed. The CDocument destructor calls DisconnectViews(). DisconnectViews() chugs through the list of views, setting CView::m_pDocument to NULL.
2. Using the views to get to the frames by calling the CView::GetParentFrame(). For example, UpdateFrameCounts(), CanCloseFrame(), and OnCloseDocument() do this.
3. Notifying the views that the document has changed and documents need to be updated. UpdateAllViews() takes care of this.

To get a feel for how these CDocument-to-CView communications work, let's take a peek at the pseudocode for CDocument::UpdateAllViews(), as shown in Listing 7-15.

Listing 7-15. CDocument::UpdateAllViews(), from DOCCORE.CPP

```
void CDocument::UpdateAllViews(CView* pSender, LPARAM lHint, CObject*
pHint)
{
    POSITION pos = GetFirstViewPosition();
    while (pos != NULL) {
        CView* pView = GetNextView(pos);
        if (pView != pSender)
```

```
    pView->OnUpdate(pSender, lHint, pHint);  
}  
}
```

It is up to the CDocument or CView to call UpdateAllViews(). If it is called from a view, usually the first argument, pSender, points to the calling view.

First, `UpdateAllViews()` gets the head of the `m_viewList` by calling `GetFirstViewPosition()`. Next, `UpdateAllView()` iterates through the list of views, and if the view is not the same as the sender, it calls `CView::OnUpdate()`. It does not call `CView::OnUpdate()` for the sender, because the caller is responsible for updating itself.

The main point to learn from `UpdateAllViews()` is that a document has to communicate with its list of views, and this is typically how it is done.

Now there is only one class we have not put under the MFC internals microscope: CView.

CView Internals

Class `CView` is the root of a view hierarchy that includes several derivatives. In this section we focus on `CView`. In Chapter 8, we take a look at some of the advanced `CView` features such as printing and print preview, as well as the `CView` derivatives.

The declaration for CView lives in AFXWIN.H. Because CView is pretty exhaustively documented, there's no need to look at the declaration. There is one undocumented data member, m_pDocument. m_pDocument is a CDocument pointer that is similar to the CDокумент::m_pDocTemplate member. It is similar because, like CDокумент::m_pDocTemplate, m_pDocument is used by CView to inform the document that it has been destroyed. The m_pDocument pointer is also used for message routing in some instances. The CView::GetDocument() member variable lets you retrieve the m_pDocument pointer.

We already saw how views are created when we looked at the CFrameWnd internals. To understand how views work, let's look at how CViews are drawn and updated in the framework.

CView Drawing

In the document/view review, remember that you need to provide a CView derivative that implements an OnDraw() member function to perform all of your drawing. Let's look at how and when your OnDraw() gets called, causing the view to be redrawn.

The usual mechanism in Windows for drawing is to handle the WM_PAINT—or in MFC, the OnPaint() message map entry for WM_PAINT. Let's look at the CView::OnPaint() handler to see how it handles redraws. Listing 7-16 contains the pseudocode for CView::OnPaint().

Listing 7-16. The CView::OnPaint() pseudocode, from VIEWCORE.CPP

```
void CView::OnPaint()
{
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

CView::OnPaint() first creates a CPaintDC, which will contain the update region for the paint. Next, OnPaint() passes the paint DC to OnPrepareDC(), and finally passes the paint DC to OnDraw().

The default CView implementations of OnPrepareDC() and OnDraw() don't really do anything; it is up to your CView derivative to provide this behavior.

We will defer covering the other interesting CView and CView derivative internals to Chapter 8. Let's review now what we have learned so far, while the CDocTemplate, CDocManager, CFrameWnd, CDocument, and CView internals are all fresh in your mind.

Document/View Internals Recap

Phew! In this chapter we've covered the internals of a surprisingly large number of classes: CDocTemplate, CDocManager, CFrameWnd (some), CDocument, and CView. To help you understand and digest everything we have uncovered, let's have a quick review.

We'll start by reiterating the various interdependencies we have discovered between all the document/view classes. Next, we review the creation of all the classes. Finally, we follow the creation of a new document (via CWinApp::OnFileNew()) through the framework to tie everything together with an example.

Document/View Interdependencies

Here are some nuggets about document/view interdependencies that we've uncovered so far:

- CWinApp contains a CDocManager pointer.
- CDocManager maintains a list of document templates. There is a document template for each type of document supported by the application. It is up to the developer to create and add these in CWinApp::InitInstance().
- Document templates bind three classes together: CView/CDocument/CFrameWnd. These classes are created from the CRuntimeClass information passed into the document template's constructor.

- The MDI-specific document template maintains a list of open documents, while the SDI-specific document template keeps only one pointer to the open document.
- Documents maintain a list of open views. They use this list to communicate updates to the views.
- Documents have a pointer to their document template. They use this for setting the title, for command routing, and for telling the document template that they have been deleted.
- Frames keep a pointer to the currently active view.
- During creation, frames get passed a CCreateContext structure, which holds copies of both the CRuntimeClass information as found in the document template as well as a pointer to the document for the frame.
- Views have a pointer to their document. This is used to inform the document that the view has been deleted (RemoveView()), and for command routing.
- Views can get to their frame window by calling CView::GetParentFrame().
- Documents can get to the frames on their views by first iterating through the view list and then calling CView::GetParentFrame().

As you work within the document/view architecture, understanding how all of these pieces fit together is extremely important. You might want to reread this list just to make sure you've got it down 100%.

Creation Information

Here's a recap of where each class is created:

- CDocManager—Created in CWinApp, stored in CWinApp::m_pDocManager.
- CDocTemplate—Created in your CWinApp::InitInstance().
- CFrameWnd—Created in CDocTemplate::CreateNewFrame().
- CDocument—Created via CDocTemplate::CreateNewDocument().
- CView—Created via CFrameWnd::OnCreate().

Putting It All Together

Looking at MFC's document/view architecture from this very low level can be like drinking from a fire hose. To give you a chance to catch your breath, let's follow the creation of a new document through the framework.

Because creating a blank document is very similar to opening a document, we'll look at both cases at the same time. Note that we follow the creation through the

default message maps and member functions. If you override any of this default behavior, the following discussion may not apply to your application.

In the Beginning . . .

The document creation process begins in your CWinApp derivative. If you generated your application using good old AppWizard, you will notice the following two lines in your CWinApp derivative's implementation file (usually the application name with .CPP appended):

```
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

These message map entries cause CWinApp::OnFileNew() to be called when the user selects New and CWinApp::OnFileOpen() to be called when the user selects Open from your application's menu.

Both of these member functions call the corresponding member function through the CWinApp CDocManager pointer, m_pDocManager.

Following New

The New and Open operations follow different paths at this point, so let's look at New and then come back to Open when we're done.

CDocManager::OnFileNew() looks through its template list (which is initialized in your CWinApp derivative's OnInitInstance() member function). If there is more than one document template to choose from, CDocManager::OnFileNew() creates a modal CNewTypeDlg and lets the user choose the type of new document to create. Once the user chooses the document type, CDocManager::OnFileNew() calls CDocTemplate::OpenDocumentFile(NULL) using the pointer returned when the user made the document type selection. Remember that this pointer was in the CDocManager's document template list, m_templateList.

Following Open

In the case of Open, CDocManager::OnFileOpen() creates a common dialog and asks the end user for the name of the file to open. After the end user has selected one, CDocManager::OnFileOpen() calls CWinApp::OpenDocumentFile(file name). CWinApp::OpenDocumentFile() calls through to CDocManager::OpenDocumentFile().

CDocManager::OpenDocumentFile() selects the correct document template based on the extension of the file name entered by the user back in CDocManager::OnFileOpen(). After finding the corresponding document template pointer, CDocManager::OnFileOpen() calls CDocTemplate::OpenDocumentFile(file name).

At this point, both Open and New are back on the same path, except New called CDocTemplate::OpenDocumentFile() with NULL and Open called it with a file name.

Figure 7-4 shows graphically where we are at this point.

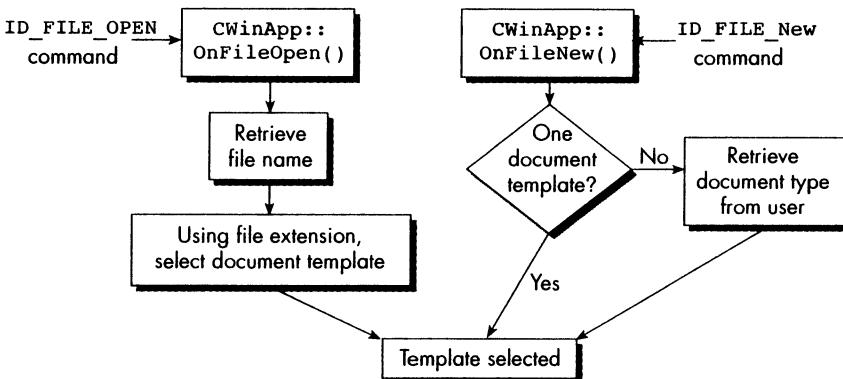


Figure 7-4. The first steps in creating/opening a document where the goal is selecting a document template

Converging Paths: CDocTemplate::OpenDocumentFile()

Depending on the type of your application (SDI or MDI), at this point either CSingleDocTemplate::OpenDocumentFile() or CMultiDocTemplate::OpenDocumentFile() is called. For this example, let's assume that CMultiDocTemplate::OpenDocumentFile() (the MDI case) is called.

First, CMultiDocTemplate::OpenDocumentFile() creates a new document by calling CDocTemplate::CreateNewDocument(), which creates a document object from the run-time information stored in the document template. Next, CMultiDocTemplate::OpenDocumentFile() creates a frame by calling CDocTemplate::CreateNewFrame(), which creates a frame object using the run-time information stored in the document template.

The creation of a new frame causes a chain of events (WM_CREATE, CFrameWnd::OnCreate(), CFrameWnd::OnCreateHelper(), CFrameWnd::OnCreateClient(), CFrameWnd::CreateView()), which eventually results in a new view being created from the view run-time information first passed to the document template and later shuffled around in a CCreateContext helper.

At this point, if a file name was specified to CDocTemplate::OpenDocumentFile(), CMyDoc::OnOpenDocument() is called to read the serialized document. Once the document is read (de-serialized), both Open and New operations follow the same code path again.

Finally, CDocTemplate::OpenDocumentFile() calls CMyDocument::OnNewDocument(), which performs any initializations.

Now all three of the document/view objects are created and ready to go. Figure 7-5 shows the flow of the last portion of the Open/New operation.

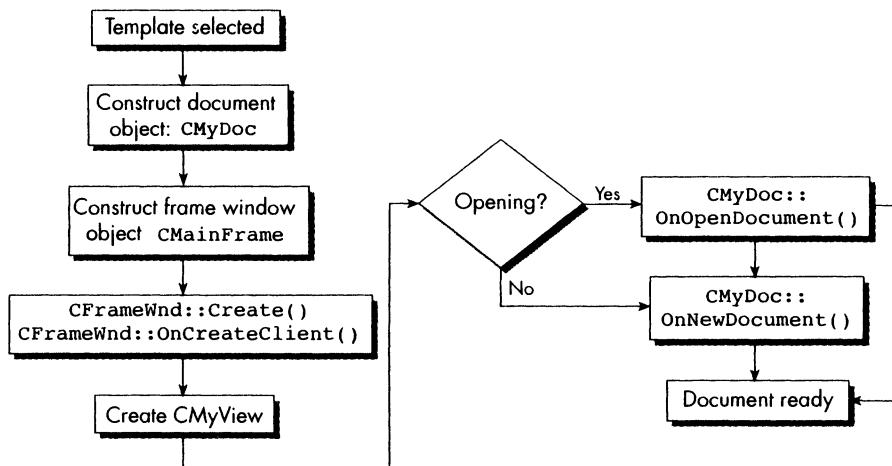


Figure 7-5. The flow of control after a template is selected

Finally, to make sure you understand this very important flow of control, Figure 7-6 shows the actual function calls along with their source files (for those of you following along at home).

Conclusion

Now that you have a pretty good understanding of what MFC's document/view architecture is and how the pieces all fit together, the next chapter dives even deeper into the architecture. Some document/view features that we explore are the undocumented CDocument helper, CMirrorFile, and advanced CView topics, such as printing, print preview, interesting derivatives, and more.

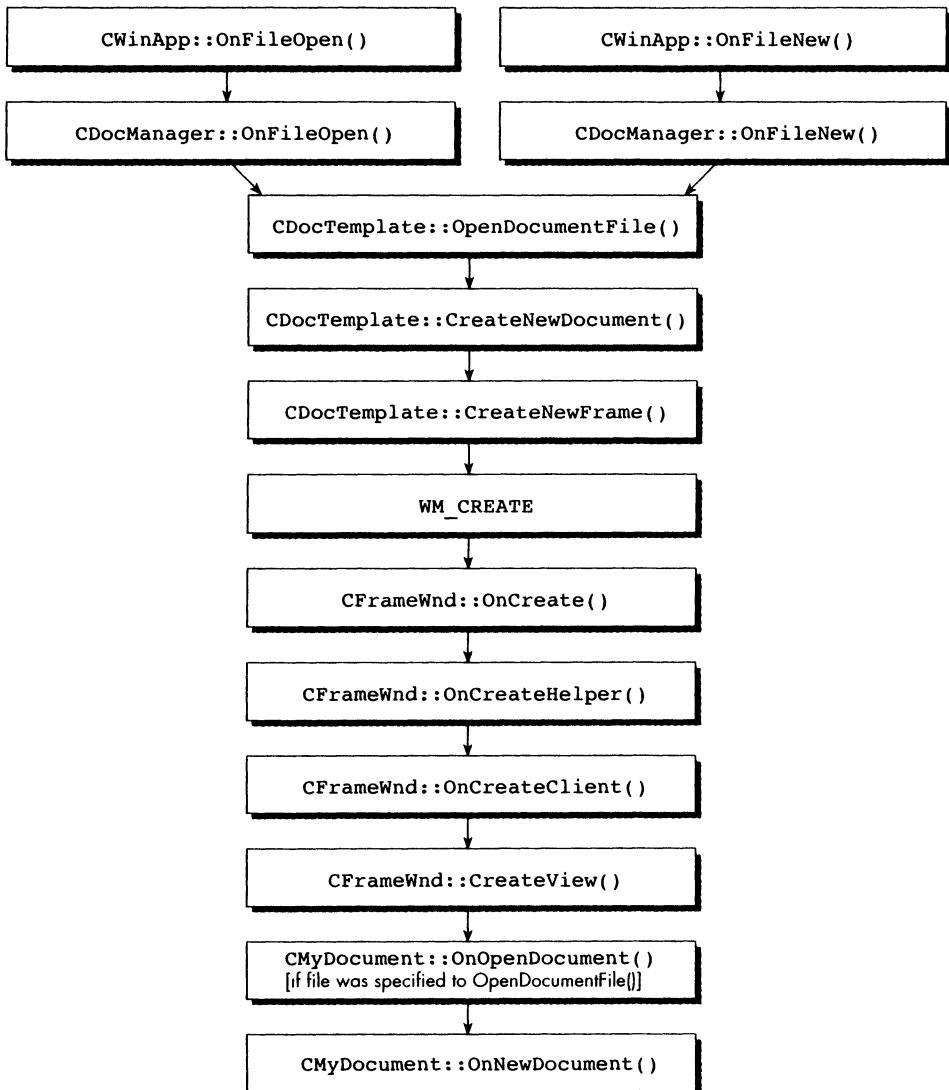


Figure 7-6. Function calls that create all of the objects for the document/view architecture

Advanced Document/View Internals

In the last chapter we looked at the document/view architecture and saw how all the pieces of the puzzle fit together: this is the first step to document/view enlightenment. In this chapter, we're going to focus more on the features that you obtain by using document/view and how MFC implements those features. Yep, you guessed it—this is the second step for doc/view enlightenment.

First we'll start off with an interesting little CDocument internal that we've discovered. After that we'll jump feet first into CView's printing and print preview support. Finally, we'll look at some of the interesting CView derivatives that provide scrolling support, form support, and even CView to Windows control interfaces.

What's the neat CDocument internal that we've discovered? Read on . . .

Mirror, Mirror, on the Wall . . .

MFC 4.0 introduced a nice improvement to the CDocument class. Two new virtual member functions, GetFile() and ReleaseFile(), let the developer specify a specialized CFile derivative (Mighty Polymorphing Power Rangers!) in a CDocument derivative.

Whenever CDocument needs to do something with a file, it calls GetFile() with the file name, file permissions, and some other arguments. GetFile() returns a CFile pointer, which can be any CFile derivative.

The fun begins when we take a peek at the default implementation of CDocument::GetFile(), as in Listing 8-1, from DOCCORE.CPP.

Listing 8-1. The implementation of CDocument::GetFile(), from DOCCORE.CPP

```
CFile* CDocument::GetFile(LPCTSTR lpszFileName, UINT nOpenFlags,
CFileException* pError)
{
```

```

CMirrorFile* pFile = new CMirrorFile;
if (!pFile->Open(lpszFileName, nOpenFlags, pError)) {
    delete pFile;
    pFile = NULL;
}
return pFile;
}

```

The implementation of GetFile() is pretty close to what you would imagine, except for the first line. What is this undocumented CMirrorFile class? Why is your document using it instead of CFile? What implications does this have to your programs? Is CMirrorFile connecting to the Microsoft Network and sending your documents to Microsoft?

There's only one way to find out.

Inside CMirrorFile

A quick search reveals that CMirrorFile is declared in our favorite header file, AFXPRIV.H. Listing 8-2 contains the declaration of this CFile derivative, from AFXPRIV.H.

Listing 8-2. CMirrorFile's declaration, from AFXPRIV.H

```

class CMirrorFile : public CFile
{
// Implementation
public:
    virtual void Abort();
    virtual void Close();
    virtual BOOL Open(LPCTSTR lpszFileName, UINT nOpenFlags,
                      CFileException* pError = NULL);
protected:
    CString m_strMirrorName;
};

```

This certainly is a pretty minimalist CFile derivative. Aside from overriding Open(), Close(), and Abort(), CMirrorFile adds a lone data member, `m_strMirrorName`.

CMirrorFile::Open()

Let's look at CMirrorFile::Open() to see what this class is up to. The implementation for CMirrorFile lives with most of CDocument's source in DOCCORE.CPP. Listing 8-3 shows the pseudocode for CMirrorFile::Open(). (You might want to sit down and catch your breath before looking, it's kind of a shocker.)

Listing 8-3. The pseudocode for CMirrorFile::Open(), from DOCCORE.CPP

```

BOOL CMirrorFile::Open(LPCTSTR lpszFileName, UINT nOpenFlags,
CFileException* pError)
{
    m_strMirrorName.Empty();
    CFileStatus status;
    if (nOpenFlags & CFile::modeCreate) {
        if (CFile::GetStatus(lpszFileName, status)){
            CString strRoot;
            AfxGetRoot(lpszFileName, strRoot);
            DWORD dwSecPerClus, dwBytesPerSec, dwFreeClus, dwTotalClus;
            int nBytes = 0;
            if (GetDiskFreeSpace(strRoot, &dwSecPerClus, &dwBytesPerSec,
                &dwFreeClus, &dwTotalClus)){
                nBytes = dwFreeClus*dwSecPerClus*dwBytesPerSec;
                if (nBytes > 2*status.m_size){
                    // get the directory for the file TCHAR
                    szPath[_MAX_PATH];
                    LPTSTR lpszName;
                    GetFullPathName(lpszFileName, _MAX_PATH, szPath,
                        &lpszName);
                    *lpszName = NULL;
                    GetTempFileName(szPath, _T("MFC"), 0,
                        m_strMirrorName.GetBuffer(_MAX_PATH+1));
                    m_strMirrorName.ReleaseBuffer();
                }
            }
        }
        if (!m_strMirrorName.IsEmpty() &&
            CFile::Open(m_strMirrorName, nOpenFlags, pError)){
            m_strFileName = lpszFileName;
            FILETIME ftCreate, ftAccess, ftModify;
            if (::GetFileTime((HANDLE)m_hFile, &ftCreate, &ftAccess,
                ftModify)) {
                AfxTimeToFileTime(status.m_ctime, &ftCreate);
                SetFileTime((HANDLE)m_hFile, &ftCreate, &ftAccess,
                    &ftModify);
            }
            DWORD dwLength = 0;
            PSECURITY_DESCRIPTOR pSecurityDescriptor = NULL;
            GetFileSecurity(lpszFileName, DACL_SECURITY_INFORMATION,
                NULL, dwLength, &dwLength);
            pSecurityDescriptor = (PSECURITY_DESCRIPTOR) new BYTE[dwLength];
            if (::GetFileSecurity(lpszFileName, DACL_SECURITY_INFORMATION,
                pSecurityDescriptor, dwLength, &dwLength)){
                SetFileSecurity(m_strMirrorName, DACL_SECURITY_INFORMATION,
                    pSecurityDescriptor);
            }
        }
    }
}

```

```

        delete[] (BYTE*)pSecurityDescriptor;
        return TRUE;
    }
    m_strMirrorName.Empty();
    return CFile::Open(lpszFileName, nOpenFlags, pError);
}

```

CMirrorFile::Open() breaks into two logical blocks. In the first block, Open() checks for the modeCreate indicating that the caller wants to create a new file or truncate an existing file. Next, Open() calls CFile::GetStatus(). GetStatus() returns nonzero if the file exists (which infers that the user wants to truncate it). In the case that the file exists, Open() then calls GetDiskFreeSpace() and determines how many bytes are available on the drive. Open() then compares this result with two times the size of the existing file. If there is enough room for two times the size of the current file name, Open() creates a temporary file and stores the name in the m_strMirrorName.

The second block of code in Open() runs only if m_strMirrorName is not empty, which implies that the contents of the first block ran and obtained a temporary file. In the second block, if there is a nonempty m_strMirrorName, Open() goes ahead and opens the mirror file using CFile::Open(). Next, Open() copies the file time and file security from the original file to the mirror file. If the second block has executed, Open() returns TRUE. If the second block does not execute, Open() calls right through to CFile::Open() returning whatever it returns.

To summarize, CMirrorFile::Open() is actually opening a different file other than the one specified if there is a write operation going on (CFile::modeCreate) and if there is a file being overwritten.

For example, in Scribble (which uses document/view), if you wrote a fresh scribble and saved it in MFCNTRNL.SCR, this code would not execute. However, if you load the MFCNTRNL.SCR file, make some changes, and then save, this code will execute and Scribble will actually write to a mirror file.

CMirrorFile::Close()

Let's take a look at CMirrorFile::Close() and see what trickery MFC is performing with our sacred document data. Listing 8-4 contains the pseudocode for CMirrorFile::Close() from DOCCORE.CPP.

Listing 8-4. The CMirrorFile::Close() implementation from DOCCORE.CPP

```

void CMirrorFile::Close()
{
    CString m_strName = m_strFileName;
    CFile::Close();
    if (!m_strMirrorName.IsEmpty()) {
        CFile::Remove(m_strName);
        CFile::Rename(m_strMirrorName, m_strName);
    }
}

```

```
    }  
}
```

First, Close() stores the name of the file in `m_strName` (a deceptively named variable) because the `CFile::Close()` call clears out `m_strFileName`.

After calling `CFile::Close()`, `CMirrorFile::Close()` checks to see if there is a mirror file in use. If there is a mirror file, `Open()` deletes the specified file and copies the mirror file over to the specified file.

CMirrorFile Conclusions

In a nutshell, `CMirrorFile` is protecting your document by saving the original and writing to a temporary. With this approach, if there is a problem writing, the original file is safe and sound. If there isn't a problem, `CMirrorFile` copies the new file over the old file and you are none the wiser (until now). `CMirrorFile` even makes sure that the original security and file creation information is copied over correctly.

Some of you out there may not like MFC doing something like this under the covers. There are also the cycles lost checking the disk space, copying the security/file time information, and copying the mirror file over the original file. If you don't like the fact that `CDocument` uses `CMirrorFile`, you can override `GetFile()` to always return a good old-fashioned vanilla `CFile` if that is your preference.

`CMirrorFile` is really the only internal of `CDocument` that warrants coverage in this "advanced" chapter. `CDocument` is where your data lives, so its internals are mostly up to the developer.

`CView` is a different story. Since its job is to display your document, there are many more interesting Windows APIs, GDI tricks, and `CView` derivatives for us to explore. We start the `CView` exploration by looking at how `CView` provides printing support.

CView Printing

`CView` is the key to MFC's printing and print preview support. By manipulating the device context that gets passed to your `CView` derivative's `OnDraw()` member function, the framework adds printing and print preview support to your applications. If you are happy with the default implementations of printing and print preview, you do not need to add any printing/print preview-specific code to your applications.

If you've ever tried to add printing to a non-MFC Windows application, you probably know that there must be a good bit of work going on under the hood. How much work? Let's see. We'll start with a quick review of some of the `CView` member functions used during printing and then investigate what makes `CView` printing tick.

CView Printing Overview

CView has several virtual functions to override for printing your document. The framework calls these functions to support printing from within the view. These functions include OnPreparePrinting(), OnBeginPrinting(), OnPrint(), OnEndPrinting(), and OnPrepareDC().

OnPreparePrinting() is called by the framework before a print job starts. OnPreparePrinting() takes a single parameter: a CPrintInfo structure. CPrintInfo contains such information as the number of pages in the document and the range of pages to be printed. You generally use OnPreparePrinting() to specify the number of pages in a document.

The framework calls OnBeginPrinting() when a printing job begins. MFC supplies a pointer to the selected printer's device context and a pointer to a CPrintInfo structure as parameters to OnBeginPrinting(). Override OnBeginPrinting() to allocate any special GDI resources for printing. You can also use OnBeginPrinting() to set the number of pages if that number depends on any special attributes of the printer device context.

The framework uses CView::OnPrint() to print a specific section of the document. If you want your printed output to appear different from your screen output, this is the function to override. You should also use this function if you have any special output that doesn't appear in your on-screen rendering (like a title page). By default, OnPrint() delegates actual printing to CView's OnDraw() function, passing the printer's device context (instead of the screen device context).

OnEndPrinting() is called by the framework after the framework is done printing. The purpose of this function is to clean up any resources allocated during OnBeginPrinting().

Finally, MFC uses OnPrepareDC() to prepare the device context if anything special needs to be done. There are three reasons why you might want to override OnPrepareDC():

1. To set the mapping mode or other characteristic of the device context for separate pages.
2. To test for the end of the document while the document is being printed. Though you normally specify the length of the document when using OnPreparePrinting(), you may not know the length of the document in advance. In such cases, you might use OnPrepareDC() to test for the end of the document.
3. To perform any other special printer functions at print time.

CView has a helper function called DoPreparePrinting(). The framework uses this function to create a device context based on the printer selected in the dialog box and to invoke the common Print dialog box.

The CPrintInfo Structure

Another important component of MFC's printing architecture is a structure called CPrintInfo. This class is documented (it's used by developers as well as by the framework). CPrintInfo maintains all the various parameters for a single print job. CPrintInfo contains an instance of CPrintDialog and functions to retrieve information from the CPrintDialog. CPrintInfo also carries general information about the print job, including whether the output is going to the printer or to the Print Preview window and the current page of the print job. As you'll see, this structure is used throughout the MFC printing cycle. Listing 8-5 shows the declaration of the CPrintInfo structure, from AFXEXT.H.

Listing 8-5. The CPrintInfo structure, as declared in AFXEXT.H

```
struct CPrintInfo // Printing information structure
{
    CPrintInfo();
    ~CPrintInfo();
    CPrintDialog* m_pPD;
    BOOL m_bPreview;
    BOOL m_bDirect;
    BOOL m_bContinuePrinting;
    UINT m_nCurPage;
    UINT m_nNumPreviewPages;
    CString m_strPageDesc;
    LPVOID m_lpUserData;
    CRect m_rectDraw;
    void SetMinPage(UINT nMinPage);
    void SetMaxPage(UINT nMaxPage);
    UINT GetMinPage() const;
    UINT GetMaxPage() const;
    UINT GetFromPage() const;
    UINT GetToPage() const;
};
```

Here's a quick look at what each field in this structure does:

- m_pPD—A pointer to a printer common dialog.
- m_bPreview—TRUE if in Preview mode, FALSE if printing.
- m_bDirect—TRUE bypasses a Print dialog, FALSE displays one.
- m_bContinuePrinting—Set to FALSE to stop printing.
- m_nCurPage—The current page.
- m_nNumPreviewPages—The number of pages to show in previewing.
- m_strPageDesc—Format string for page number display.

- `m_lpUserData`—pointer to a user-created struct.
- `m_rectDraw`—A rectangle that defines the current usable page area.
- `CPrintInfo()`—Creates a `CPrintDialog` and stores it in `m_pPD`. Also initializes the data members and some fields in the `m_pPD->m_pd` `PRINTDLG` structure such as min and max page.
- `SetMinPage()`—Sets the min page field in the `m_pPD->m_pd` `PRINTDLG` structure.
- `SetMaxPage()`—Sets the max page field in the `m_pPD->m_pd` `PRINTDLG` structure.
- `GetMinPage()`, `GetMaxPage()`, `GetFromPage()`, `GetToPage()`—Retrieve the relevant fields from the `m_pPD->m_pd` `PRINTDLG` structure.

We'll see these `CPrintInfo` fields in action in the next section, when we look under the hood of `CView` printing.

CView Printing Internals

Printing kicks off when the user selects File/Print. The `CView` message map entry is this:

```
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
```

Let's start with `OnFilePrint()` and follow the printing trail through `CView`. Note: You will find most of the `CView` printing code in the MFC source file `VIEWPRNT.CPP`.

CView::OnFilePrint()

`OnFilePrint()` is an extremely long member function, so we'll have to cut it into easily digested pieces. Let's start with the first part of `OnFilePrint()`, which prepares to do the printing. We'll look at the actual printing loop in the next chunk.

Listing 8-6 contains the first piece of `OnFilePrint()`, from `VIEWPRNT.CPP`.

Listing 8-6. The first piece of CView::OnFilePrint(), from VIEWPRNT.CPP

```
void CView::OnFilePrint()
{
    CPrintInfo printInfo;
    // **omitted - OnFilePrint() checks the command line to see if it
    // should display a CPrintDialog or not (m_bDirect data member
    // is set.
    if (OnPreparePrinting(&printInfo)) {
        //**omitted - Will get filename if user is printing to file
```

```

// by displaying a CFileDialog.
CString strTitle;
CDocument* pDoc = GetDocument();
strTitle = pDoc->GetTitle();
// setup the printing DC and DOCINFO
DOCINFO docInfo;
docInfo.cbSize = sizeof(DOCINFO);
docInfo.lpszDocName = strTitle;
CDC dcPrint;
dcPrint.Attach(printInfo.m_pPD->m_pd.hDC);
dcPrint.m_bPrinting = TRUE;
OnBeginPrinting(&dcPrint, &printInfo);
dcPrint.SetAbortProc(_AfxAbortProc);
AfxGetMainWnd()->EnableWindow(FALSE);
CPrintingDialog dlgPrintStatus(this);
// *omitted - CPintingDialog setup
dlgPrintStatus.ShowWindow(SW_SHOW);
dlgPrintStatus.UpdateWindow();
// ... To be continued in next chunk

```

OnFilePrint() first declares a CPintInfo on the stack and constructs it. Inside its constructor, CPintInfo allocates a common Print dialog and assigns it to the m_pPD variable. CPintInfo sets the minimum number of pages in the common Print dialog box to 1 and the maximum to 0xffff (65,565) pages—so that there are some numbers in those fields. CPintInfo also initializes its other member variables: m_nCurPage is set to 1, m_lpUserData is set to NULL, m_bPreview is set to FALSE, and m_bContinuePrinting is set to TRUE.

Once OnFilePrint() has constructed a CPintInfo structure, OnFilePrint() calls OnPreparePrinting(). This is a virtual function, so control is passed to the derived class's version (if the function is overridden). If you use AppWizard to generate your program, CMYView::OnPreparePrinting() calls CView::DoPreparePrinting().

DoPreparePrinting() creates the printer DC by displaying the CPintDialog that is stored in the CPintInfo structure. After the user has selected the various printing options from the common Print dialog, the printer DC is stored in pInfo->m_pPD->m_pd.hDC.

After calling OnPreparePrinting(), OnFilePrint() gets the title of the document and creates a DOCINFO structure. The DOCINFO structure contains the document name and the name of the output device. This structure will be used later as an argument to the Win32 StartDoc() API.

Next, OnFilePrint() creates a local CDC on the stack and attaches it to the DC handle that was created in DoPreparePrinting() and is stored in the PRINTDIALOG structure. Once the CDC has been created, OnFilePrint() sends the DC along with the CPintInfo structure to CView::OnBeginPrinting(). The default implementation

of `OnBeginPrinting()` does nothing, but if you ever want to modify the DC that will be used for printing, overload `OnBeginPrinting()` and go for it.

After calling `OnBeginPrinting()`, `OnFilePrint()` starts to get serious about printing. It sets an abort procedure for the DC, `_AfxAbortProc()`, and then disables the main window. Then `OnFilePrint()` creates, initializes, and displays a `CPrintingDialog`.

What's a `CPrintingDialog`? The `CPrintingDialog` is a small printing status dialog that displays the page number and also lets the user cancel the printing operation. Figure 8-1 shows the `CPrintingDialog` for our favorite MFC application, Scribble.

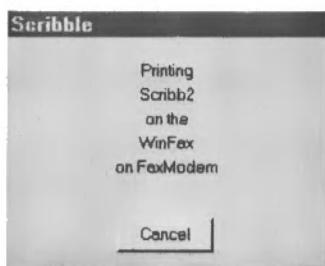


Figure 8-1. Scribble's CPrintingDialog

If the user wants to cancel printing, `CPrintingDialog::OnCancel()` sets a global abort flag to TRUE. During printing the `_AfxAbortProc()` checks the flag and, if TRUE, aborts the print job.

At this point in `OnFilePrint()`, a `CPrintInfo` has been created, the `CPrintDialog` has been shown, a printing DC has been created, and the `CPrintingDialog` has been shown. Now we're (finally) ready to print! Let's look at the remaining code for `OnFilePrint()` and see how it goes about using all of the objects created in the first section of code.

What If You Want to Change the "Printing Dialog"?

Let's face it, `CPrintingDialog` is a very boring printing dialog. Having a neat progress indicator or a picture of a printer printing would be much fancier. So how do you go about changing the dialog that `OnFilePrint()` uses? Unfortunately, the answer isn't simple. Unlike the convenient `CDocument::GetFile()`, there is no `CView::GetPrintDlg()`. This is one of those times in MFC when you have to use some brute strength. The following steps show how to shoehorn in your own printing dialog:

1. In your `CView` derivative, add a `CMyView::OnFilePrint()`.
2. Change the message map to call your `OnFilePrint()` instead of `CView`'s.
3. Copy all of the code from `CView::OnFilePrint()` into your `OnFilePrint()`.
4. Modify the code segment around the `CPrintingDialog` calls to use your `CDialog` derivative instead of `CPrintingDialog`.

5. Be sure that your printing dialog Cancel is similar to CPrintingDialog::OnCancel().
6. Make sure that you have changed all occurrences of CPrintingDialog, especially in the bottom of OnFilePrint(), where the dialog is updated.

Hopefully, future versions of MFC will add more customization routines, so that these not-so-elegant hacks won't be necessary.

The CView::OnFilePrint() Printing Loop

Listing 8-7 contains the pseudocode for the last part of OnFilePrint().

Listing 8-7. The remaining pseudocode for CView::OnFilePrint()

```

// start document printing process
if (dcPrint.StartDoc(&docInfo) == SP_ERROR)
    // **omitted, some error handling that cleans and returns
int nStep = (nEndPage >= nStartPage) ? 1 : -1;
nEndPage = (nEndPage == 0xffff) ? 0xffff : nEndPage + nStep;
// begin page printing loop
BOOL bError = FALSE;
for (printInfo.m_nCurPage = nStartPage; printInfo.m_nCurPage != nEndPage;
    printInfo.m_nCurPage += nStep){
    OnPrepareDC(&dcPrint, &printInfo);
    // check for end of print
    if (!printInfo.m_bContinuePrinting)
        break;
    TCHAR szBuf[80];
    wsprintf(szBuf, strTemp, printInfo.m_nCurPage);
    dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PAGENUM, szBuf);
    // set up drawing rect to entire page (in logical coordinates)
    printInfo.m_rectDraw.SetRect(0, 0, dcPrint.GetDeviceCaps(HORZRES),
        dcPrint.GetDeviceCaps(VERTRES));
    dcPrint.DPtoLP(&printInfo.m_rectDraw);
    // attempt to start the current page
    if (dcPrint.StartPage() < 0){
        bError = TRUE;
        break;
    }

    OnPrint(&dcPrint, &printInfo);
    if (dcPrint.EndPage() < 0 || !_AfxAbortProc(dcPrint.m_hDC, 0)){
        bError = TRUE;
        break;
    }
} //end page for loop here.
// cleanup document printing process
if (!bError)
    dcPrint.EndDoc();

```

```

    else
        dcPrint.AbortDoc();
    AfxGetMainWnd()->EnableWindow();      // enable main window

    OnEndPrinting(&dcPrint, &printInfo);    // clean up after printing
    dlgPrintStatus.DestroyWindow();
    dcPrint.Detach();      // will be cleaned up by CPrintInfo destructor
}
}

```

In this section of `OnFilePrint()`, first `CDC::StartDoc()` is called to kickstart the Windows printing engine. Next, `OnFilePrint()` calculates a step (-1 for 1 page, or 1 for multiple pages), determines the end page, and sets `bError` to FALSE.

`OnFilePrint()` then enters the “for” loop that actually does the printing. This “for” loop follows these steps for each page in the document:

1. Calls `OnPrepareDC()`, whose default implementation doesn’t do anything.
2. Updates the page number on the `CPrintingDialog`.
3. Sets the `CPrintInfo.m_rectDraw` rectangle to the size of the page.
4. Calls `CDC::StartPage()` to signify that a new page is about to be printed.
5. Calls `CView::OnPrint()`, with the printer DC and `CPrintInfo` structure as arguments. `CView::OnPrint()` calls `CView::OnDraw()`. At this point, your `CView` derivative draws to the DC and generates the printer output.
6. Calls `EndPage()`.

After every page has been printed, `OnFilePrint()` calls `EndDoc()`, re-enables the window, and calls `OnEndPrinting()`. The default implementation of `OnEndPrinting()` does nothing. If you allocated something in a printing override, this is a good virtual to override and release whatever you allocated.

Finally, `OnFilePrint()` destroys the `CPrintingDialog` printer status dialog and detaches from the DC.

To recap, `OnFilePrint()` drives the printing process by

- preparing to print,
- printing each page, and
- cleaning up.

Throughout the printing process, overrideable virtual functions provide opportunities for the developer to customize the printing process. Figure 8-2 shows the MFC printing process in a flow diagram.

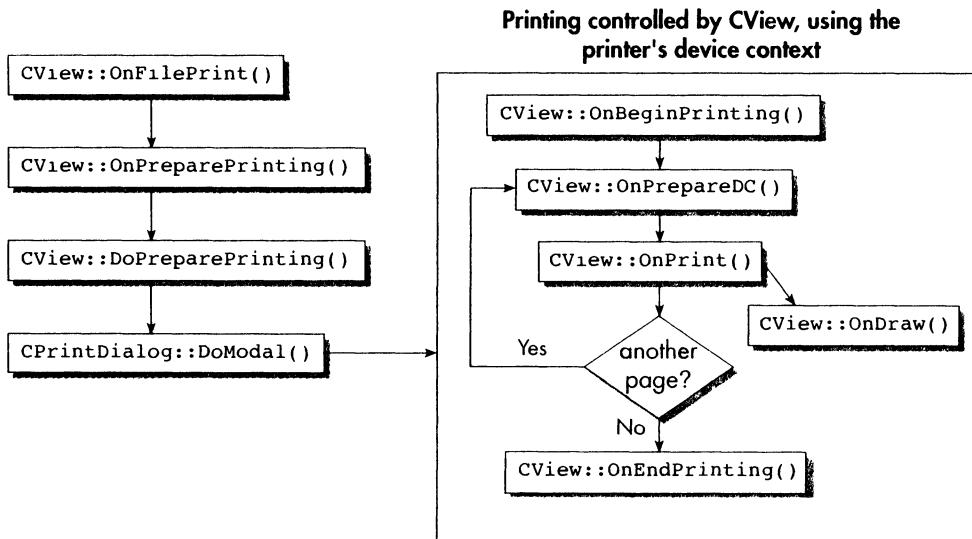


Figure 8-2. MFC printing process flow diagram

CView print previewing works on similar principles, but because it has the complex task of mimicking the printed page on your screen, it has to jump through more hoops.

Inside CView Print Preview Support

Like printing, print previewing ultimately uses your `CMYView::OnDraw()` routine to generate the displayed output. But unlike printing, there's much more going on under the hood. If you look at the MFC Print Preview window as shown in Figure 8-3, many questions should come to mind:

- How does MFC replace your normal window with the Print Preview window?
- How does it draw those fancy little page outlines?
- How does it support zooming and scrolling?
- How does it simulate the printer output on the display?
- Where does that toolbar come from?

We'll answer these questions and more. It turns out that there is a veritable gold mine of undocumented classes and helpers buried deep within the CView print preview support. As always, we'll be sure to shine floodlights on them as they come up.

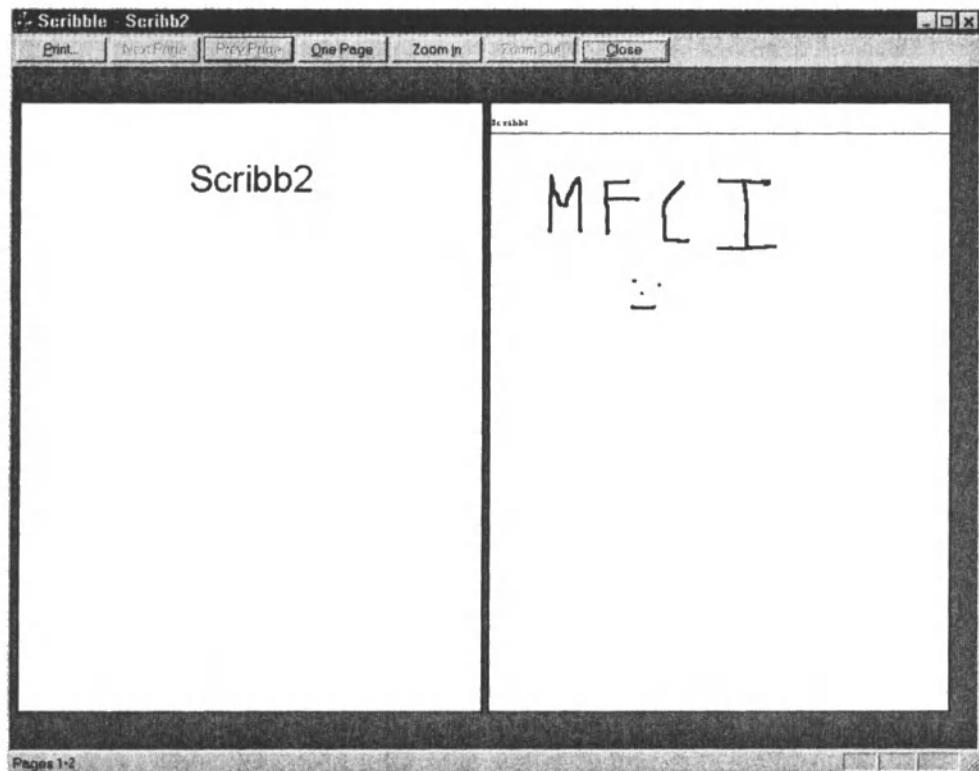


Figure 8-3. Example of print preview

CView::OnFilePrintPreview()

Like printing, print preview gets started via a message map entry for the File/Print Preview menu item:

```
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
```

Let's look at CView::OnFilePrintPreview() to see what happens next. Listing 8-8 contains the pseudocode for this routine. Note: All of the CView print preview support lives in VIEWPREV.CPP.

Listing 8-8. CView::OnFilePrintPreview(), from VIEWPREV.CPP

```
void CView::OnFilePrintPreview()
{
    CPrintPreviewState* pState = new CPrintPreviewState;
    if (!DoPrintPreview(AFX_IDD_PREVIEW_TOOLBAR, this,
```

```

RUNTIME_CLASS(CPreviewView), pState)) {
    delete pState; // preview failed to initialize, delete State now
}
}

```

In comparison to `OnFilePrint()`, `OnFilePrintPreview()` certainly is tiny. It creates a `CPrintPreviewState` object from the heap and then passes it along to `DoPrintPreview()`. `DoPrintPreview()` is also being called with the resource ID for the Print Preview toolbar (`AFX_IDD_PREVIEW_TOOLBAR`), a this pointer, and `RUNTIME_CLASS(CPreviewView)`. If you know your MFC hierarchy, the next-to-last argument should cause the “Undocumented MFC class” alarms to go off in your head. Next, let’s look at `CPrintPreviewState`, and `DoPrintPreview()`; we’ll save the intriguing `CPreviewView` for dessert.

CPrintPreviewState

`CPrintPreviewState` is a helper structure (like `CCreateContext` and `CPrintInfo`) that stores information about the print preview as it gets passed through the `CView` print preview architecture.

`CPrintPreviewState` is declared in `AFXEXT.H` and includes these fields:

- `nIDMainPane`—The resource ID of the main pane to be hidden.
- `hMenu`—A handle to a saved menu.
- `dwStates`—State bits for control bar visible states.
- `pViewActiveOld`—Stores the previously active view during preview.
- `hAccelTable`—A saved accelerator table.

We’ll see more details about the various `CPrintPreviewState` fields as we dig deeper. Now let’s look at `DoPrintPreview()`, which is called by `OnFilePrintPreview()`.

Inside DoPrintPreview()

`CView::DoPrintPreview()` takes a plethora of arguments. Listing 8-9 contains the pseudocode for the member function, from `VIEWPREV.CPP`.

Listing 8-9. The `CView::DoPrintPreview()` pseudocode, from `VIEWPREV.CPP`

```

BOOL CView::DoPrintPreview(UINT nIDResource, CView* pPrintView,
                           CRuntimeClass* pPreviewViewClass,
                           CPrintPreviewState* pState)
{
    CFrameWnd* pParent = (CFrameWnd*)AfxGetThread()->m_pMainWnd;
}

```

```

CCreateContext context;
context.m_pCurrentFrame = pParent;
context.m_pCurrentDoc = GetDocument();
context.m_pLastView = this;
// Create the preview view object
CPreviewView* pView = (CPreviewView*)pPreviewViewClass->CreateObject();
pView->m_pPreviewState = pState;           // save pointer
pParent->OnSetPreviewMode(TRUE, pState);    // Take over Frame Window

// Create the toolbar from the dialog resource
pView->m_pToolBar = new CDialogBar;
if (!pView->m_pToolBar->Create(pParent,
                                  MAKEINTRESOURCE(nIDResource), CBRS_TOP,
                                  AFX_IDW_PREVIEW_BAR)){
    TRACE0("Error: Preview could not create toolbar dialog.\n");
    return FALSE;
}
pView->m_pToolBar->m_bAutoDelete = TRUE;    // automatic cleanup
if (!pView->Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
                    CRect(0,0,0,0), pParent, AFX_IDW_PANE_FIRST, &context)) {
    TRACE0("Error: couldn't create preview view for frame.\n");
    return FALSE;
}
pState->pViewActiveOld = pParent->GetActiveView();
CView* pActiveView = pParent->GetActiveFrame()->GetActiveView();
pActiveView->OnActivateView(FALSE, pActiveView, pActiveView);
pView->SetPrintView(pPrintView);
pParent->SetActiveView(pView);   // set active view - even for MDI
// update toolbar and redraw everything
pView->m_pToolBar->SendMessage(WM_IDLEUPDATECMDUI, (WPARAM)TRUE);
pParent->RecalcLayout();          // position and size everything
pParent->UpdateWindow();
return TRUE;
}

```

DoPrintPreview() first stores a pointer to the main frame window in pParent. Next, it creates a local CCreateContext structure and populates its fields.

Next, DoPrintPreview() creates a CPreviewView instance using dynamic creation using the CRuntimeClass pointer argument. DoPrintPreview() then calls CFrameWnd::OnSetPreviewMode() and creates a toolbar based on the toolbar resource argument.

After creating and storing the toolbar in the CPreviewView pointer, DoPrintPreview() creates the view and makes it the active view. When setting the CPreviewView to the active view, DoPrintPreview() saves the previously active view in the CPaintPreviewState structure.

Once the CPreviewView and toolbar are created and activated, DoPrintPreview() updates them and finally returns TRUE.

We've discovered a good deal about print preview by now. One interesting point is how DoPrintPreview() sets the active view in the main frame to switch from the regular MDI or SDI window to the Print Preview view. You can use this technique in your MFC applications if you ever want to have a "full window mode" or want to produce a similar effect. To learn more about how it works, check out CFrameWnd::OnSetPreviewMode().

At this point, most of our print preview questions are still unanswered. All of the clues so far point to one suspect: the undocumented class CPreviewView.

CPreviewView: An Undocumented Print Preview CView Derivative

Inside MFC's most top-secret header file, AFXPRIV.H, we found the declaration for CPreviewView, which reveals a great deal about the class. Listing 8-10 contains the abbreviated declaration.

Listing 8-10. The CPreviewView declaration, from AFXPRIV.H

```
class CPreviewView : public CScrollView
{
    DECLARE_DYNCREATE(CPreviewView)
// Constructors
public:
    CPreviewView();
    BOOL SetPrintView(CView* pPrintView);
// Attributes
protected:
    CView* m_pOrigView;
    CView* m_pPrintView;
    CPreviewDC* m_pPreviewDC; // Output and attrib DCs Set, not created
    CDC m_dcPrint;           // Actual printer DC
// Operations *omitted
// Overridables *omitted
// Implementation * some omitted
public:
    virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
protected:
    afx_msg void OnPreviewClose();
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnDraw(CDC* pDC);
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

```

afx_msg BOOL OnEraseBkgnd(CDC* pDC);
afx_msg void OnNextPage();
afx_msg void OnPrevPage();
afx_msg void OnPreviewPrint();
afx_msg void OnZoomIn();
afx_msg void OnZoomOut();
void DoZoom(UINT nPage, CPoint point);
void SetScaledSize(UINT nPage);
CSize CalcPageDisplaySize();
CPrintPreviewState* m_pPreviewState; // State to restore
CDialogBar* m_pToolBar; // Toolbar for preview
struct PAGE_INFO {
    CRect rectScreen; // screen rect (screen device units)
    CSize sizeUnscaled; // unscaled screen rect (screen device units)
    CSize sizeScaleRatio; // scale ratio (cx/cy)
    CSize sizeZoomOutRatio; // scale ratio when zoomed out (cx/cy)
};
PAGE_INFO* m_p PageInfo; // Array of page info structures
PAGE_INFO m_pageInfoArray[2]; // Embedded array for the default
// implementation
BOOL m_bPageNumDisplayed; // Flags whether or not page number has yet
// been displayed on status line
UINT m_nZoomOutPages; // number of pages when zoomed out
UINT m_nZoomState;
UINT m_nMaxPages; // for sanity checks
UINT m_nCurrentPage;
UINT m_nPages;
int m_nSecondPageOffset; // used to shift second page position
HCURSOR m_hMagnifyCursor;
CSize m_sizePrinterPPI; // printer pixels per inch
CPoint m_ptCenterPoint;
CPrintInfo* m_pPreviewInfo;
DECLARE_MESSAGE_MAP()
};


```

Holy cow! CPreviewView is definitely a heavyweight undocumented class. Notice that it is a CScrollView derivative. We haven't covered this class, but as the name implies, CScrollView provides scrolling support. When you zoom in on a print preview, you'll notice that it displays scroll bars: that's the CScrollView inheritance at work.

CPreviewView is chock full of really interesting internals. Here's what the more interesting data members are used for:

- **m_pOrigView**—A pointer to the previously active view.
- **m_pPrintView**—A pointer to the view for “printing.”

- m_pPreviewDC—A pointer to a CPreviewDC. (What's a CPreviewDC? We'll cover it soon.)
- m_dcPrint—A pointer to a printer DC.
- PAGE_INFO—A convenience structure that defines the dimensions of one of the preview pages.
- m_pPageInfo—A pointer to a PAGE_INFO structure.
- m_pageInfoArray—An array of two-page info structures.
- m_bPageNumDisplayed—A flag that indicates if the page number has been displayed yet or not.
- m_nZoomOutPages—Stores the number of pages to be displayed on one print preview view.
- m_nZoomState—Can be one of four zoom states (set via SetZoomState()):
 - ZOOM_OUT—The default, shows a smaller than 1:1 ratio.
 - ZOOM_MIDDLE—A zoom level between out and in.
 - ZOOM_IN—Shows a larger than 1:1 ratio.
 - ZOOM_OFF—No zooming.
- m_nMaxPages—Specifies the number of pages. This is always 2 for CPreviewView.
- m_nCurrentPage—The current page being displayed.
- m_nPages—The number of pages. Can be either 1 or 2 in the default CPreviewView.
- m_nSecondPageOffset—Specifies the left coordinate of where the second preview page is drawn.
- m_hMagnifyCursor—A handle to the magnifying glass cursor.
- m_sizePrinterPPI—Stores the dimensions of a page on the printer retrieved by calling GetDeviceCaps(LOGPIXELSX).
- m_ptCenterPoint—Never used.
- m_pPreviewInfo—A pointer to a CPrintInfo structure that is used to store printing information.

Some of these data members raise some interesting rhetorical questions about the class:

- What is ZOOM_MIDDLE and why is it never used?
- Why have the m_ptCenterPoint if it's never used?

Let's start our CPreviewView exploration by digging around in the SetPrintView() member function. If you'll recall, this member function is called in CView::DoPrintPreview() (see Listing 8-9).

Inside CPreviewView::SetPrintView()

SetPrintView() is a very important member for CPreviewView. It takes care of initializing the print preview view and preparing it to start the print previewing process. Keep that in mind as you review the pseudocode in Listing 8-11, from VIEWPREV.CPP.

Listing 8-11. The pseudocode for CPreviewView::SetPrintView(), as found in VIEWPREV.CPP

```
BOOL CPreviewView::SetPrintView(CView* pPrintView)
{
    m_pPrintView = pPrintView;
    m_pPreviewInfo = new CPrintInfo;
    m_pPreviewInfo->m_pPD->SetHelpID(AFX_IDD_PRINTSETUP);
    m_pPreviewInfo->m_pPD->m_pd.Flags |= PD_PRINTSETUP;
    m_pPreviewInfo->m_pPD->m_pd.Flags &= ~PD_RETURNDC;
    m_pPreviewInfo->m_bPreview = TRUE; // signal that this is preview
    m_pPreviewDC = new CPreviewDC; // must be created before any
    if (!m_pPrintView->OnPreparePrinting(m_pPreviewInfo))
        return FALSE;
    m_dcPrint.Attach(m_pPreviewInfo->m_pPD->m_pd.hDC);
    m_pPreviewDC->SetAttribDC(m_pPreviewInfo->m_pPD->m_pd.hDC);
    m_pPreviewDC->m_bPrinting = TRUE;
    m_dcPrint.m_bPrinting = TRUE;
    m_dcPrint.SaveDC(); // Save pristine state of DC
    HDC hDC = ::GetDC(m_hWnd);
    m_pPreviewDC->SetOutputDC(hDC);
    m_pPrintView->OnBeginPrinting(m_pPreviewDC, m_pPreviewInfo);
    m_pPreviewDC->ReleaseOutputDC();
    ::ReleaseDC(m_hWnd, hDC);
    m_dcPrint.RestoreDC(-1); // restore to untouched state
    // Get Pixels per inch from Printer
    m_sizePrinterPPI.cx = m_dcPrint.GetDeviceCaps(LOGPIXELSX);
    m_sizePrinterPPI.cy = m_dcPrint.GetDeviceCaps(LOGPIXELSY);
    m_nPages = m_pPreviewInfo->m_nNumPreviewPages;
    m_nZoomOutPages = m_nPages;
    SetScrollSizes(MM_TEXT, CSize(1, 1)); // initialize mapping mode only
    if (m_pPreviewInfo->GetMaxPage() < 0x8000 &
        m_pPreviewInfo->GetMaxPage() - m_pPreviewInfo->GetMinPage() <=
        32767U)
        SetScrollRange(SB_VERT, m_pPreviewInfo->GetMinPage(),
                      m_pPreviewInfo->GetMaxPage(), FALSE);
    else
        ShowScrollBar(SB_VERT, FALSE);
    SetCurrentPage(m_pPreviewInfo->m_nCurPage, TRUE);
    return TRUE;
}
```

SetPrintView() first sets the value of `m_pPrintView` to the document's view. Remember, the document's view actually does the rendering in `OnDraw()`. `CPreviewView` has to keep a copy of the document's live view because `CPreviewView` uses the view's `OnDraw()` function to render the data on the preview screen. Next, `SetPrintView()` creates a new `CPrintInfo` object. From our look at `CView` printing, we discovered that the `CPrintInfo` constructor creates a `CPrintDialog`. After creating the `CPrintInfo` object, `SetPrintView()` sets some flags. The most important flag tells the `CPrintDialog` not to return a DC. We'll see why in a second.

After creating the `CPrintInfo` object, `SetPrintView()` creates a `CPreviewDC` object. What's a `CPreviewDC`, you ask? It's another undocumented MFC class! (Didn't we tell you that print preview was full of them?) MFC's `CDC` class contains two handles to device contexts: `m_hDC` and `m_hAttribDC`. The `m_hDC` member represents the device context for the output, and the `m_hAttribDC` is used for information purposes (a.k.a. the "read-only" DC)—usually these device contexts represent the same thing, the screen or the printer.

`CPreviewDC` maintains two different device contexts: one for the screen (`m_hDC`) and one for the printer (`m_hAttribDC`). `CPreviewDC` is implemented this way because print preview has to draw something as though it would appear on the printer, but be able to translate it so it appears correctly on the screen. `SetPrintView()` sets up the `CPreviewDC` accordingly—setting the `m_hDC` to the view's `hDC` and setting `m_hAttribDC` to the printer's `hDC`. To learn more about how `CPreviewDC` handles displaying the printer output on-screen, check out the MFC header `AFXPRIV.H` and the source file `DCPREV.CPP`.

After setting up the DCs to their proper print preview state, `SetPrintView()` initializes the `m_sizePrinterPPI` data member, as well as `m_nPages` and `m_nZoomOutPages`. Finally, `SetPrintView()` initializes the scroll bars in the scroll view and sets the current page.

Drawing the Print Preview

So far in our print preview exploration, the trail has been pretty clear. First we got the scent of `CPreviewView` in `OnFilePrintPreview()` and followed it all the way to `SetPrintView()`. But we still haven't seen where the actual drawing of the print preview happens.

If you'll look back at Listing 8-9 (`CView::DoPrintPreview()`), you'll notice that this routine "forces" an update of the frame window that contains the freshly created `CPreviewView`. Like any other MFC view, this will cause `CPreviewView`'s `OnDraw()` member function to be called.

`CPreviewView::OnDraw()` is a whopper of a member function, so we will not even attempt to represent it here. We do recommend that you open up `VIEWPREV.CPP` while you follow this discussion of `OnDraw()`. First, `OnDraw()` creates two pens:

`rectPen` is for the box that is drawn to represent the page; `shadowPen` is used to draw the shadow around the page to give it a fancy 3-D effect.

Next, `OnDraw()` enters a “for” loop for each page. In this “for” loop, it takes the following steps for each page:

1. Draws the page outline and shadow using the paint DC that is passed as argument to `OnDraw()`. It draws the rectangle with a series of `MoveTo()`/`LineTo()` calls. Then the “for” loop calls `FillRect()` to draw in the white color of the pages.
2. Displays the page number by calling `OnDisplayPageNumber()`.
3. Once the fresh piece of paper has been drawn, `OnDraw()` calls `m_pPrintView->OnPrint()` with the `CPreviewDC`, which generates the print preview output.

After the “for” loop has iterated through all of the pages to be previewed, `OnDraw()` frees the pens it created.

Print Preview Wrapup

To recap:

- MFC print preview is largely the job of two undocumented classes: `CPreviewView` and `CPreviewDC`.
- To kick off print preview, the `CView` basically swaps itself for a `CPreviewView` and gets out of the way until previewing is over.
- From that point on, `CPreviewView` takes care of the Print Preview toolbar, zooming, and other aspects of print preview.
- `CPreviewDC` lends a helping hand by providing the means to mimic on-screen what the printer will output.

For More Information

We could go on for chapters and chapters with more details of `CPreviewView` and `CPreviewDC`, but we have to proceed to other `CView` points of interest. For readers who would like to continue exploring, here are some questions to get you on your way and stimulate some serious brainwaves:

- See if you can figure out how print preview zooming works. (Hint: Start at `CPreviewView::DoZoom()` and check out `SetScaledSize()`.)
- How does the page number get drawn and what class draws it?
- Why does `CPreviewDC` need its own versions of the `DrawText()` and `TextOut()` family of GDI functions?

- How would you modify MFC's print preview to be able to display four pages, the way Microsoft Word does?

Let's continue our advanced hunting of CView internals by looking at some of the CView derivatives. First up is CScrollView.

CView Derivatives: CScrollView

You probably remember CScrollView from the first time you took the MFC Scribble tutorial. It's the CView derivative that lets you easily add scrolling to your views. To use CScrollView, you derive your application-specific view from CScrollView and then call SetScrollSizes() to let CScrollView know the size of your "logical" view.

CScrollView takes the size information and manipulates the DC that is passed to your OnDraw() such that your application automatically supports scrolling. CScrollView also has an interesting, often-overlooked feature that automatically redraws your view so that it always fits the client area. MFC refers to this as "scale to fit" mode.

This is all fine and dandy, but how does CScrollView go about accomplishing what used to take days to add to your Windows application? To find out, let's see how CScrollView works.

Note: This is a good time to brush up on your Petzold GDI mapping mode and viewport concepts, because CScrollView uses them pretty heavily. We'll assume that you have some understanding of these APIs, we'll explain the more obscure ones as we encounter them.

How CScrollView Works

To figure out how CScrollView is implemented, let's first look at the implementation details from the CScrollView declaration. CScrollView is declared in AFXWIN.H, and its implementation lives in VIEWSCRL.CPP. Listing 8-12 contains an abbreviated CScrollView declaration with the implementation members highlighted.

Listing 8-12. The CScrollView declaration, from VIEWSCRL.CPP

```
class CScrollView : public CView
{
    DECLARE_DYNAMIC(CScrollView)
// Constructors ** omitted.
// Attributes **omitted.
// Operations **omitted.
```

```

// Implementation **some omitted
protected:
    int m_nMapMode;
    CSize m_totalLog;
    CSize m_totalDev;
    CSize m_pageDev;
    CSize m_lineDev;
    BOOL m_bCenter;
    void CenterOnPoint(CPoint ptCenter);
    void ScrollToDevicePosition(POINT ptDev); .
protected:
    void UpdateBars();
    BOOL GetTrueClientSize(CSize& size, CSize& sizeSb);
    void GetScrollBarSizes(CSize& sizeSb);
    void GetScrollBarState(CSize sizeClient, CSize& needSb,
                           CSize& sizeRange, CPoint& ptMove, BOOL bInsideClient);
public:
    virtual void CalcWindowRect(LPRECT lpClientRect,
        UINT nAdjustType = adjustBorder);
    virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
    virtual BOOL OnScroll(UINT nScrollCode, UINT nPos,
        BOOL bDoScroll = TRUE);
    virtual BOOL OnScrollBy(CSize sizeScroll, BOOL bDoScroll = TRUE);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
    afx_msg void OnVScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
DECLARE_MESSAGE_MAP()
};


```

Here's a quick overview of the CScrollView implementation members in the order that they appear in the declaration:

CScrollView implementation data members:

- **m_nMapMode**—In the SetScrollSizes(), you can specify a mapping mode for your application. The default is MM_NONE. CScrollView defines a “custom” mapping mode (MM_SCALETOFIT) that we'll look at later. The user can specify any mapping mode except for the MM_ISOTROPIC and ANISOTROPIC. These mapping modes are so flexible that CScrollView could not be written to anticipate all of the possibilities.
- **m_totalLog**—The total size of the view in logical coordinates. This value is passed into CScrollView through the SetScrollSizes() member function.
- **m_totalDev**—The total size of the view in device coordinates.
- **m_pageDev**—The size of a page in device coordinates.

- `m_lineDev`—The size of a line in device coordinates.
- `m_bCenter`—This data member is used by `CPreviewView` to enter the view in the client window.

CScrollView implementation member functions:

- `CenterOnPoint()`—Centers the view on a point. This function is called by `CPreviewView`. (See `m_bCenter`.)
- `ScrollToDevicePosition()`—As the name implies, this routine actually scrolls the view. It accomplishes this by both updating the scroll bars using `::SetScrollPos()` and calling `::ScrollWindow()`.
- `UpdateBars()`—This helper is called by `CScrollView` at initialization and when the window is sized. The job of `UpdateBars()` is to hide, display, and initialize scroll bars based on the information returned by `GetScrollBarState()`.
- `GetTrueClientSize()`—This member is used to determine if the client is large enough to have scroll bars. `GetTrueClientSize()` is called only by `UpdateBars()`.
- `GetScrollBarSizes()`—Determines the width and height of a scroll bar, taking into consideration the window styles and border widths. Both `GetScrollBarState` and `GetTrueClientSize()` use the size of the scroll bars in their calculations.
- `GetScrollBarState()`—Retrieves information about the state of the view needed by `CScrollView`. For example, it calculates if the view needs scroll bars, what the current scroll position is, and so on.
- `CalcWindowRect()`—Calculates the size of the window rectangle, taking into account scroll bar sizes and other window dressings.
- `OnPrepareDC()`—The key to the `CScrollView/CView` interaction. We'll look at this member in detail a little later.
- `OnScroll()`—Both of the scroll bar message handlers, `OnHScroll()` and `OnVScroll()`, call `OnScroll()`, passing along the information from the scroll message. `OnScroll()` decodes the scroll message and determines the amount to scroll based on the page and line sizes. Once the amount to scroll is calculated, `OnScroll()` calls `OnScrollBy()`.
- `OnScrollBy()`—This member function checks that the amount to scroll is within the range of the scroll bars and the logical size of the view. If everything is cool, `OnScrollBy()` moves the scroll bars by calling `::SetScrollPos()` and then calls `::ScrollWindow()`.
- `OnSize()`—If the `CScrollView` is not in scale-to-fit mode, `OnSize()` calls `UpdateBars()`. If the `CScrollView` is in scale-to-fit mode, `OnSize()` calls `SetScaleToFitSize()`.
- `OnHScroll()`—A message handler that calls `OnScroll()`.
- `OnVScroll()`—A message handler that calls `OnScroll()`.

If you've been paying close attention, you may have detected some duplication of code in these implementation helpers. Both ScrollToDevicePosition() and OnScrollBy() eventually do the same thing: scroll using SetScrollPos() and ScrollWindow(). Why have this duplication of code?

The reason is that OnScrollBy() is called via user interaction, so it performs lots of checks to make sure that everything is valid. On the other hand, ScrollToDevicePosition() is called by the developer using CScrollView through the documented ScrollToPosition() member function. In certain situations, the developer might not want the checks that are enforced by the code in OnScrollBy(). To make everyone happy, MFC provides two ways to achieve basically the same thing.

Let's start our CScrollView exploration by looking at the member function that you have to call in your CScrollView derivative to initialize the CScrollView::SetScrollSizes().

CScrollView::SetScrollSizes()

Listing 8-13 contains the pseudo-code for CScrollView::SetScrollSizes(), from VIEWSCRL.CPP

Listing 8-13. The CScrollView::SetScrollSizes() pseudocode, from VIEWSCRL.CPP

```
void CScrollView::SetScrollSizes(int nMapMode, SIZE sizeTotal, const SIZE&
    sizePage, const SIZE& sizeLine)
{
    int nOldMapMode = m_nMapMode;
    m_nMapMode = nMapMode;
    m_totalLog = sizeTotal;

    {
        CWindowDC dc(NULL);
        dc.SetMapMode(m_nMapMode);
        m_totalDev = m_totalLog;
        dc.LPtoDP((LPPOINT)&m_totalDev);
        m_pageDev = sizePage;
        dc.LPtoDP((LPPOINT)&m_pageDev);
        m_lineDev = sizeLine;
        dc.LPtoDP((LPPOINT)&m_lineDev);
        if (m_totalDev.cy < 0)
            m_totalDev.cy = -m_totalDev.cy;
        if (m_pageDev.cy < 0)
            m_pageDev.cy = -m_pageDev.cy;
        if (m_lineDev.cy < 0)
            m_lineDev.cy = -m_lineDev.cy;
    }
    if (m_pageDev.cx == 0)
```

```

m_pageDev.cx = m_totalDev.cx / 10;
if (m_pageDev.cy == 0)
    m_pageDev.cy = m_totalDev.cy / 10;
if (m_lineDev.cx == 0)
    m_lineDev.cx = m_pageDev.cx / 10;
if (m_lineDev.cy == 0)
    m_lineDev.cy = m_pageDev.cy / 10;
if (m_hWnd != NULL){
    UpdateBars();
    if (nOldMapMode != m_nMapMode)
        Invalidate(TRUE);
}

```

First, SetScrollSizes() initializes m_nMapMode to the new mapping mode provided. In addition, m_totalLog is initialized to the sizeTotal argument. SetScrollSizes() then creates a CWindowDC on the stack, and after setting the mapping mode, it uses the DC to calculate the device coordinates for the total size, page size, and line size.

After converting the logical coordinates to device coordinates, SetScrollSizes() checks to make sure that the user has provided nondefault values. If the defaults are specified (0 is the default), SetScrollSizes() sets the page size to 1/10 the total size and the line size to 1/10 the page size, or 1/100 of the total size.

Finally, SetScrollSizes() calls UpdateBars() and Invalidate() if needed. UpdateBars() sets up the scroll bars based on the new sizes and Invalidate() causes a repaint if the mapping mode has changed.

Block Those DCs!

At first glance, the extra curly braces right before the CWindowDC creation and right after the line device calculations in Listing 8-13 may surprise you. Why does Microsoft have extra braces in here? The answer has to do with one of the scarcest of Windows resources, the device context. Because device contexts are a pretty valuable resource, MFC wants to minimize their use, even if it means freeing them a couple of code lines earlier than the default.

Wrapping a stack-created DC (like the CWindowDC in SetScrollSizes()) with curly braces creates another stack frame. When the compiler hits the end of the nested stack frame, it will go ahead and destroy the stack variables, thus freeing the DC that was created on the stack. Without these extra curly braces, the DC would not have been freed until the very end of SetScrollSizes(). With the call to UpdateBars() and Invalidate(), this could take longer than you think!

The lesson is that if you are using stack-created DCs in your classes, you should consider "blocking" them off with a new stack frame to minimize the time that they are around.

One frequent block of code you will see in our CScrollView travels is this:

```
//...
CWindowDC dc(NULL);
dc.SetMapMode(m_nMapMode);
dc.LPtodP(...);
//...
```

Because your CScrollView derivative could be drawing the view using a variety of different mapping modes, CScrollView has to be careful to convert from logical coordinates to device coordinates and vice versa. Using a DC for the calculation is the best way to do the conversion.

As mentioned in the member overview, CScrollView and CView interact through one override, CScrollView::OnPrepareDC(). Let's look at this override next.

CScrollView::OnPrepareDC()

If you'll recall earlier in this chapter and in the previous chapter, we mentioned that CView::OnPrepareDC() should be overridden to provide any customizations to the device context used for display or printing. Guess what? CScrollView does exactly that. Let's look at the pseudo code for CScrollView's OnPrepareDC() override and see what the heck it's doing with the DC. Listing 8-14 contains the relevant code from VIEWSCRL.CPP.

Listing 8-14. The CScrollView::OnPrepareDC() pseudocode, from VIEWSCRL.CPP

```
void CScrollView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    switch (m_nMapMode){
        case MM_SCALETOFIT:
            pDC->SetMapMode(MM_ANISOTROPIC);
            pDC->SetWindowExt(m_totalLog); // window is in logical
                                            // coordinates
            pDC->SetViewportExt(m_totalDev);
            break;
        default:
            pDC->SetMapMode(m_nMapMode);
            break;
    }
    CPoint ptVpOrg(0, 0); // assume no shift for printing
    if (!pDC->IsPrinting()) {
        ptVpOrg = -GetDeviceScrollPosition();
        // **omitted, some m_bCenter stuff.
    }
    pDC->SetViewportOrg(ptVpOrg);
    CView::OnPrepareDC(pDC, pInfo); // For default Printing behavior
}
```

Before we explain what `OnPrepareDC()` does, remember that it will always be called before `OnDraw()` with the DC that is passed to `OnDraw()`.

First, `OnPrepareDC()` sets the mapping mode. If the `CScrollView` is in “scale-to-fit” mode, `OnPrepareDC()` sets the mapping mode to ANISOTROPIC and then manipulates the window and viewport extents such that the contents of the view fit in the client window. The end result from these mapping mode manipulations is that when you draw in `OnDraw()`, GDI automatically “scales” the drawing to fit the client window. When `CScrollView` is not in scale-to-fit mode, `OnPrepareDC()` calls `SetMapMode()` with the mapping mode specified by the user in `SetScrollSizes()`.

Next comes the trick to the `CScrollView` implementation, so watch carefully. `OnPrepareDraw()` creates a `CPoint` on the stack called `ptVpOrg` (shorthand for viewport origin). `ptVpOrg` is initialized to 0,0. If the view is not printing, `OnPrepareDC()` subtracts the negative of the current scroll position from the viewport origin. Next, `OnPrepareDC()` calls `CDC::SetViewportOrg()` with the `ptVpOrg` (which will still be 0,0 if we’re printing) and then calls the overridden `CView::OnPrepareDC()`.

If you’re not a hard-core GDI programmer, this may seem mysterious to you. Basically, `OnPrepareDC()` is manipulating the device context such that your `OnDraw()` code offsets its drawing by the amount of the scroll. Thus, your `OnDraw()` code is none the wiser that it is drawing at an offset.

For example, if you call `SetPixel(10,10)`, and the scroll bars are at 10,5, the point is actually drawn at pixel 0,5. The origin is added to any GDI coordinates. For this example: $10+(-10)=0$ and $10+(-5)=5$.

CScrollView Recap

Once you understand what the `CScrollView` members are used for and what `SetScrollSizes()` and `OnPrepareDC()` are up to, you know 90 percent of the `CScrollView` implementation. Remember that `SetScrollSizes()` initializes the mapping mode, page size, and line size. It then calls `UpdateBars()` to actually create and display the scroll bars if needed. Also, remember that `OnPrepareDC()` causes your view to scroll by manipulating the viewport origin of the DC. Finally, the actual scrolling is performed by `OnScrollBy()` and `ScrollToDevicePosition()`, which use `ScrollWindow()`.

For More Information

Here are some suggestions to get you on your way to more `CScrollView` discoveries:

- Start with `OnSize()` and see if you can figure out how the view is redrawn to fit the new client window size in scale-to-fit mode.
- Can you deduce the intention of the `m_bCenter` member function by looking at its usage in `OnPrepareDC()` and `GetDeviceScrollPosition()`?

- Browse through UpdateBars():
 - Any guesses as to the use of m_bInsideUpdate?
 - Any ideas what the WM_RECALCPARENT message is doing?

Now that we understand what's going on inside CScrollView, let's take a look at one of its derivatives, CFormView.

CFormView: Forms in a View

CFormView lets you create a view that is composed of controls created from a dialog template. This class is particularly helpful if you have lots of form-based user interfaces. The developer benefits from having the features and simplicity of a dialog template merged with the power of the document/view architecture. The result is a form that scrolls and supports DDX/DDV and can even be used in a splitter window (which we'll cover in the next chapter). Unfortunately, CFormView does not support automatic printing or print preview, because it is up to the developer to decide what the printed version of the form should look like.

Using a CFormView is similar to using CDialog. First, you create a dialog template with some controls using the VC++ resource editor. Next, you create a CFormView derivative and attach it to the dialog template by passing the resource ID of the template to the CFormView constructor.

One interesting difference in using a CFormView versus a normal CView is that you do not need to override OnDraw(). The controls in the dialog template take care of displaying themselves, so you do not need to get involved.

Generally, most of your CFormView code will be performing DDX/DDV with the controls. (We covered DDX/DDV back in Chapter 6; you may want to flip back there now and review.)

Under the Hood of CFormView

CFormView is declared in AFXEXT.H and implemented in VIEWFORM.CPP. The declaration for CFormView doesn't add too much to CScrollView except for some CScrollView overrides and a couple of new data members. Here's a synopsis of the data members and some of the more interesting overrides:

- m_lpszTemplateName—The dialog template resource name.
- m_pCreateContext—A pointer to the CCreateContext (see Chapter 6). CFormView does not use this directly, but passes it through the framework in the OnCreate() override.

- `m_hWndFocus`—In `OnSetFocus()`, `CFormView` sets the focus to this window handle unless it is invalid. `OnActivateView()` and `OnActivateFrame()` both maintain the data member.
- `OnDraw()`—Does nothing! The Windows controls will draw themselves, so there is nothing for `CFormView` to do in `OnDraw()`.
- `PreTranslateMessage()`—`CFormView` has to do some message-routing trickery to get all of the messages meant for the controls, view, and frame to the right destination. All of this takes place in `PreTranslateMessage()`.
- `SaveFocusControl()`—Sets `m_hWndFocus` to the control with focus by setting it to the result of `::GetFocus()`.
- `OnActivateFrame()`—Calls `SaveFocusControl()` when the view goes inactive to make sure that the control with focus retains control when focus comes back. This takes place in `OnSetFocus()`.
- `OnActivateView()`—Sets the `m_hWndFocus` variable too.
- `OnCreate()`—Passes `m_pCreateContext` through the framework in the `lpCreateParams` field of a `LPCREATESTRUCT` structure pointer.
- `OnSetFocus()`—If `m_hWndFocus` points to a valid window, `OnSetFocus()` sets focus to that control. If `m_hWndFocus` does not point to a valid control, `OnSetFocus()` defers the focus handling to Windows.

One `CFormView` member function warrants more investigation than a cursory overview: `CFormView::Create()`.

Inside `CFormView::Create()`

This `CWnd::Create()` override does 90% of the work for `CFormView`. Let's look at how `Create()` goes about creating and initializing the `CFormView`. Listing 8-15 contains the pseudocode for `CFormView::Create()`.

Listing 8-15. The `CFormView::Create()` pseudocode

```
BOOL CFormView::Create(LPCTSTR , LPCTSTR , DWORD dwRequestedStyle,
                      const RECT& rect, CWnd* pParentWnd,
                      UINT nID, CCreateContext* pContext)
{
    m_pCreateContext = pContext;
    VERIFY(AfxDeferRegisterClass(AFX_WNDCOMMCTL_REG));
    CREATESTRUCT cs;
    memset(&cs, 0, sizeof(CREATESTRUCT));
    if (dwRequestedStyle == 0)
        dwRequestedStyle = AFX_WS_DEFAULT_VIEW;
```

```

cs.style = dwRequestedStyle;
if (!PreCreateWindow(cs))
    return FALSE;
if (!CreateDlg(m_lpszTemplateName, pParentWnd))
    return FALSE;
m_pCreateContext = NULL;
ModifyStyle(WS_BORDER|WS_CAPTION, dwRequestedStyle &
(WS_BORDER|WS_CAPTION));
ModifyStyleEx(WS_EX_CLIENTEDGE, cs.dwExStyle & WS_EX_CLIENTEDGE);
SetDlgItemID(nID);
CRect rectTemplate;
GetWindowRect(rectTemplate);
SetScrollSizes(MM_TEXT, rectTemplate.Size());
if (!ExecuteDlgInit(m_lpszTemplateName))
    return FALSE;
SetWindowPos(NULL, rect.left, rect.top, rect.right - rect.left,
rect.bottom - rect.top, SWP_NOZORDER|SWP_NOACTIVATE);
if (dwRequestedStyle & WS_VISIBLE)
    ShowWindow(SW_NORMAL);
return TRUE;
}

```

Create() first stores the CCreateContext argument in m_pCreateContext so that it can pass it along in OnCreate(). Next, Create() makes sure that the Windows common controls are registered.

In the next block of code (from the CREATESTRUCT down to the PreCreateWindow() call), Create() calls PreCreateWindow to determine the extended style (which will be stored in cs.dwExStyle) specified by the developer. After determining the extended style, Create() calls CWnd::CreateDlg(). CreateDlg() loads the dialog template from the application's resources and calls CWnd::CreateDlgIndirect() (see Chapter 5 for the details of this member function). CreateDlg() results in a modeless dialog being created.

After calling CreateDlg(), Create() NULLs out m_pCreateContext, because it is no longer needed. Next, Create() modifies the normal and extended styles of the window. For the normal style, it makes sure that the border and caption are set as requested in dwRequestedStyle. For the extended style, it makes sure that WS_EX_CLIENTEDGE is set correctly as specified in the CREATESTRUCT passed to PreCreateWindow(). Create() then sets the control identifier of the window to the nID argument. Next, a call to SetScrollSizes() is made to initialize the CScrollView to use MM_TEXT mapping mode and to create a logical view the size of the dialog template.

In addition to loading the dialog template from the application's resources, Create() calls CWnd::CreateDlg(). CreateDlg() is a CWnd member function that calls CreateDlgIndirect to create a modeless dialog. Now that the controls have been created and the scroll bars are in place if necessary, Create() calls ExecuteDlgInit() (remember this from

Chapter 5?) to initialize the controls in the form if any initial values were specified in the resource file.

Finally, Create() forces the size of the view to match the rect argument and then calls ShowWindow() if WS_VISIBLE was specified in the PreCreateWindow() call. Once the CFormView is created, the only real interesting implementation detail is DDX/DDV, which we've covered in detail already. UpdateData(FALSE) is called in CFormView::OnInitialUpdate() to initialize the controls in the form. After that, DDX/DDV behaves as if the CFormView were a dialog.

CFormView Recap

Class CFormView is a very thin CScrollView derivative that creates a modeless dialog from a dialog resource template and provides a CView interface to the dialog.

In the next mini-hierarchy of CView derivatives we look at, the concept is similar, but instead of providing a CView interface for a dialog, the classes provide a CView interface for a single control.

Another CView Derivative: CCtrlView

CCtrlView is a CView derivative that enables a Windows control to be a view. CCtrlView is a base class that you should never use directly in your applications, but you probably will use one of its popular derivatives: CEditView, CListView, CTreeView, and CRichEditView.

In this section, first we'll look at how CCtrlView does the control-to-view mapping and then we'll take a peek inside one of the CCtrlView derivatives to see how it works with the CCtrlView interface.

Version Note

CCtrlView was introduced with MFC 4.0. Prior to MFC 4.0, the only control view was CEditView, which was derived directly from CView. It was the addition of the three new CView control classes (CListView, CTreeView, and CRichEditView) that spurred the creation of a base class to define the CView-to-control interface. In MFC 4.0 CEditView's derivation was transparently changed from CView to CCtrlView.

CCtrlView: How It Works

CCtrlView is declared in AFXWIN.H and its implementation lives in VIEWCORE.CPP. The declaration is pretty short, so we've gone ahead and reproduced it here in Listing 8-16.

Listing 8-16. The CCtrlView declaration, from AFXWIN.H

```

class CCtrlView : public CView
{
    DECLARE_DYNCREATE(CCtrlView)
public:
    CCtrlView(LPCTSTR lpszClass, DWORD dwStyle);
// Attributes
protected:
    CString m_strClass;
    DWORD m_dwDefaultStyle;
// Overrides
    virtual void OnDraw(CDC*);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
// Implementation ** some debug stuff omitted
protected:
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()
};

```

CCtrlView really doesn't do much. Its primary goal in life is to provide an interface to its derivatives that makes them fit in with CView. With this in mind, here's a rundown of what each member does:

- **m_strClass**—This CString data member contains the name of the window class for the control that is being “view-ized.” m_strClass is set in the CCtrlView constructor and is passed in through a constructor argument.
- **m_dwDefaultStyle**—The default style for the view class. This data member is also passed via a CCtrlView constructor argument.
- **CCtrlView()**—The constructor sets m_strClass and m_dwDefaultStyle.
- **OnDraw()**—Never gets called! It calls ASSERT(FALSE) to ensure that this is the case. See **OnPaint()**.
- **PreCreateWindow()**—Sets the lpszClass of the CREATESTRUCT to the m_strClass data member so that the control is created. Also manipulates the CREATESTRUCT style field to reflect the value stored in m_dwDefaultStyle.
- **OnPaint()**—Calls Default(), which sends the previous message (WM_PAINT) to DefWindowProc(). CCtrlView does this because the controls can paint themselves and do not need any help.

OK, we admit that CCtrlView really isn't as exciting as it seems. The exciting part of CCtrlView is the implementation of these interfaces by its derivatives.

Let's look at one of the newer CCtrlView derivatives as an example: CTreeView.

CTreeView: An Example Control View/ CCtrlView Derivative

CTreeView is declared in AFXCVIEW.H and implemented in VIEWCMN.CPP/AFXCVIEW.INL. The declaration for CTreeView is in Listing 8-17.

Listing 8-17. The CTreeView declaration, from AFXCVIEW.H

```
class CTreeView : public CCtrlView
{
    DECLARE_DYNCREATE(CTreeView)
// Construction
public:
    CTreeView();
// Attributes
public:
    CTreCtrl& GetTreeCtrl() const;
protected:
    void RemoveImageList(int nImageList);
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    afx_msg void OnDestroy();
    DECLARE_MESSAGE_MAP()
};
```

As you can tell from Listing 8-17, CTreeView basically provides implementations for the CCtrlView members we discussed and adds a couple of new functions. The new member functions are these:

- `GetTreeCtrl()`—Returns a CTreeCtrl reference that the class user can use to make direct CTreeCtrl calls and manipulate the tree control.
- `RemoveImageList()`—An internal helper routine that clears out image lists in the control. This helper is called in `OnDestroy()`, where the LSVIL_NORMAL and LVSIL_STATE image lists are cleared out of the control. These are the identifiers given to the image lists containing large icons and state images.

CTreeView::CTreeView()

Here's the implementation of the CTreeView() constructor from AFXCVIEW.INL:

```
CTreeView::CTreeView() :
    CCtrlView(WC_TREEVIEW, AFX_WS_DEFAULT_VIEW) { }
```

All the constructor does is pass the name of the class, which is #defined to WC_TREEVIEW, to the base class, CCtrlView. CTreeView() also passes AFX_WS_DEFAULT_VIEW, the default window style bits, to the base class.

CTreeView::PreCreateWindow()

All PreCreateWindow() does is make sure that the Windows common controls are initialized before calling CCtrlView::PreCreateWindow().

CTreeView::GetTreeCtrl()

Finally, GetTreeCtrl() is implemented as the following:

```
CTreeCtrl& CTreeView::GetTreeCtrl() const
{
    return *(CTreeCtrl*)this;
}
```

Why does this work? It works because CTreeCtrl has the same binary representation as CWnd and since CTreeView is derived from CWnd this is safe. In other words, a CTreeCtrl is only a CWnd (from a binary standpoint) and since CTreeView is a CWnd it can also be a CTreeCtrl.

CCtrlView Wrapup

As you can tell, the CCtrlView derivatives are really thin, especially CListView and CTreeView. CRichEditView has some interesting internals for supporting OLE, but those are beyond the scope of this chapter. CEditView also has some interesting internals, but most of them arise from the limitations of the classic edit control.

For More Information

If you're interested in exploring the other CCtrlView derivatives, here's where you can find their declarations and implementations:

	<i>Declaration</i>	<i>Implementation</i>
CEditView	AFXEXT.H	AFXEXT.INL/VIEWEDIT.CPP
CListView	AFXVIEW.H	AFXVIEW.INL/VIEWCMN.CPP
CRichEditView	AFXRICH.H	AFXRICH.INL/VIEWRICH.CPP

Here are some questions to get you going:

- CEditView—This CCtrlView derivative supports printing. How?
- CEditView is serializeable. Why would you want to serialize a view?
- CListView—How does CListView support owner-drawn list views?

- CRichEditView—Explore the CRichEditDoc and CRichEditCntrItem classes. Can you figure out what they are used for and why they are needed?
- What is the undocumented CReObject used for?

Conclusion

Now that we've seen both the basic and advanced details of MFC's document/view architecture, let's move on and examine some of MFC's enhanced user-interface classes, such as control bars and splitter windows. Splitter windows depend on many of the CView details covered here, so be prepared for a CView quiz!

MFC's Enhanced User-Interface Classes

So far we've examined a variety of MFC user-interface classes. Besides the Windows encapsulations like CWnd, CDialog, and all of the control classes, there's a category of user-interface classes that do not encapsulate existing Windows objects. We call these the enhanced user-interface classes. Because CView is still fresh in your mind (hopefully!) from the last chapter, we'll start this chapter with splitter windows, which work very closely with views.

Next, we'll look at the CControlBar family of classes, which includes dialog bars, toolbars, and status bars. We'll be sure to show you how MFC's nifty dockable toolbars work, too. We'll finish up this chapter by looking at the MFC CMiniFrame CFrameWnd derivative.

Because the implementation for most of these classes lives entirely in MFC (with the exception of CToolBar and CStatusBar with MFC 4.0), there are tons of really interesting internals to explore. Let's get started by looking at what makes CSplitterWnd tick.

CSplitterWnd: MFC Splitter Windows

CSplitterWnd is one of the most complicated and thus confusing of all MFC classes. CSplitterWnd has to be complicated in order to provide such an advanced interface. After reading this section, you should finally understand how CSplitterWnd works and be more comfortable using it in your MFC applications.

If you've used any of the popular Microsoft Windows products, you've probably encountered some type of splitter window. Static splitter windows help divide a window logically into two resizable areas. The Windows 95 Explorer and the Visual C++ bitmap editor are examples of static splitter windows. Dynamic splitter windows

let the user arbitrarily divide a window into multiple views on the same data. The Visual C++ code editor and Microsoft Word use dynamic splitters.

Before we get too involved in CSplitterWnd, let's dissect a splitter window.

The Anatomy of a Splitter Window

Figure 9-1 shows the key components of a splitter window.

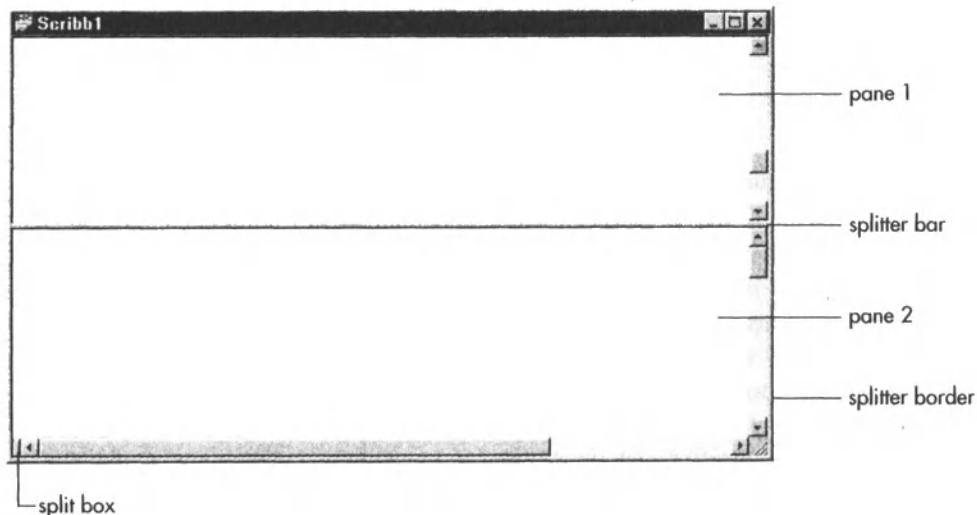


Figure 9-1. Splitter window components

A splitter window, like a real-world window, contains panes. In the figure there are only two panes, but in MFC, there can be either two or four panes. Panes are usually CViews, but CSplitterWnd is flexible enough to allow panes to be any CWnd derivative.

You create a pane by “grabbing” the split box (see the figure) and dragging it to the position where you want the splitter bar to appear. When there are four panes and thus two splitter bars, the point where the bars meet is called a splitter intersection (not pictured).

We'll discuss the mysterious splitter border when we look at how MFC draws all of the splitter bar components.

Splitter windows internally maintain the panes in terms of rows and columns. For example, the panes in Figure 9-1 are two rows; if the split were vertical, there would be two columns. Finally if there were both a horizontal and a vertical split, there would be two columns and two rows.

Because splitter windows use views as panes, when the user changes the document data, the results are automatically updated in each pane via the CDocument::UpdateAllViews()

mechanism. If you use CWnd derivatives as panes, this synchronization will be up to you.

Finally, note that splitter windows have their own scroll bars. Splitter windows need their own scroll bar logic because the scroll bars often affect two panes at once. For example, in Figure 9-1, the bottom horizontal scroll bar scrolls both pane 1 and pane 2.

Now that you've seen the parts of a splitter window, let's take a brief refresher course in how to use CSplitterWnd.

Refresher: How to Use CSplitterWnd

As previously mentioned, there are two ways to use CSplitterWnd: dynamic and static.

To create a dynamic splitter window in your MFC application, you need to follow these steps:

1. Add a CSplitterWnd data member to your child frame derivative. (this will vary depending on whether you are using MDI or SDI). For example:

```
CSplitterWnd m_wndSplitter;
```

2. In the CMyChildFrame::OnCreateClient() handler, add a call to CSplitterWnd::Create(). Create() takes several arguments. The first argument is a pointer to the parent frame (usually "this"). The second and third arguments specify the maximum number of rows and columns. The fourth argument specifies a minimum allowable pane size. Finally, the fifth argument is a pointer to a CCreateContext (see Chapter 7).

The following example creates a splitter window with a maximum of two rows and two columns. The minimum pane size is 10 by 10, and the context pointer is passed through without change.

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT , CCreateContext*
    pContext)
{
    return m_wndSplitter.Create(this, 2, 2, CSize(10, 10), pContext);
}
```

In MFC, dynamic splitters are limited to two by two, so the row and column arguments are mostly used to restrict the rows and columns by specifying 1.

3. Pass the RUNTIME_CLASS information for your frame window to the CDocTemplate constructor:

```
CMultiDocTemplate * pDocTemplate =
new CMultiDocTemplate(IDR_MYAPPTYPE,
RUNTIME_CLASS(CMyDocClass), RUNTIME_CLASS(CMyChildFrameClass),
```

```
RUNTIME_CLASS(CMyViewClass));
AddDocTemplate(pDocTemplate);
```

That's all you need to add dynamic splitter windows to your MFC applications! When the user creates a split, MFC will automatically take care of filling the splitter with the panes (views) that are wired together with the splitter window in the document template (in this example, CMyViewClass).

To use static splitter windows, you call `CreateStatic()` instead of `Create()`. It is also up to the developer to create the views for each static window, because MFC does not know the type of view to create. You create new panes by calling `CSplitterWnd::CreateView()`.

Static splitter windows do not have the same two-by-two restrictions that dynamic splitter windows have. You may be asking yourself: "If `CSplitterWnd` can statically draw any number of rows and columns, then why can't it do this dynamically?" Great question! We'll investigate this when we start looking at the `CSplitterWnd` internals.

Here's an example that creates a "grid" of four-by-four static splitter panes:

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT , CCreateContext* pContext)
{
    int nRow;
    int nCol;
    if (!m_wndSplitter.CreateStatic(this,4,4))
        return FALSE;
    for (nRow = 0; nRow < 4; nRow++)
        for (nCol = 0; nCol < 4; nCol++)
            m_wndSplitter.CreateView(nRow,nCol,RUNTIME_CLASS(CMyView),
                                     CSize(10,10), pContext);
    return TRUE;
}
```

Once the views are created, you can call `CSplitterWnd` member functions like `SetColumnInfo/SetRowInfo()` to specify the size of the static splitters.

Enough overview! Let's see how MFC implements this really neat user-interface gadget.

Inside `CSplitterWnd`

Looking back at the refresher, you may realize that `CSplitterWnd` is one of the easiest MFC classes to use. Even dialogs and controls require you to create some resources and such. As a pretty hip MFC internalist (you've made it to Chapter 9!), this should alert you to the fact that `CSplitterWnd` must be doing massive amounts of work under the hood to compensate for the simplicity of its interface.

There are so many interesting implementation details in CSplitterWnd that we're going to have to approach it a little differently from the other MFC classes. Usually we dive right into the source code and start tracing through the important functions. With CSplitterWnd, we'll take a top-down approach and look at the following areas of the MFC CSplitterWnd implementation:

- Declaration—We'll start by looking at the implementation-specific portions of the CSplitterWnd declaration.
- Initialization—Next, we'll look at the Create(), CreateStatic(), and CreateView() member functions to see what's happening when you call these from your MFC applications.
- Pane management—After seeing how CSplitterWnd is initialized, we'll examine how it goes about managing the panes of the splitter.
- Drawing—Next, we'll go a little deeper and see how MFC actually draws the different parts of a splitter window, including the splitter box, splitter bars, splitter intersection, and all of the other splitter window components.
- Hit testing/tracking—After seeing how a splitter window is drawn, we'll look at another interesting implementation detail: hit testing and tracking. CSplitterWnd uses hit testing to figure out which splitter window component the user is trying to interact with. Hit testing is a technique that you can apply to many of your MFC/Windows situations. Tracking is used to implement the user interface where the user "grabs" the splitter bar and drops it in the desired location.
- Performing a split—Finally, we'll tie it all together by following a split through the framework and seeing exactly how all of these CSplitterWnd features work together.

Throughout the CSplitterWnd discussion, we won't differentiate much between static and dynamic splitters, since they share 99 percent of the same implementation code. Whenever there is something specific to dynamic or static splitter, we'll be sure to point it out.

The CSplitterWnd Declaration

CSplitterWnd is declared in AFXEXT.H. The implementation-specific aspects of the declaration are reproduced for you in Listing 9-1.

Listing 9-1. The Abbreviated CSplitterWnd declaration, from AFXEXT.H

```
class CSplitterWnd : public CWnd
{
    DECLARE_DYNAMIC(CSplitterWnd)
// Construction ** omitted.
// Attributes **omitted.
```

```

// Operations
public:
    virtual void RecalcLayout();      // call after changing sizes
// Overridables **some omitted.
protected:
    enum ESplitType { splitBox, splitBar, splitIntersection, splitBorder };
    virtual void OnDrawSplitter(CDC* pDC, ESplitType nType,
        const CRect& rect);
    virtual void OnInvertTracker(const CRect& rect);
public:
    virtual BOOL CreateScrollBarCtrl(DWORD dwStyle, UINT nID);
public:
// high level command operations - called by default view implementation
    virtual BOOL CanActivateNext(BOOL bPrev = FALSE);
    virtual void ActivateNext(BOOL bPrev = FALSE);
    virtual BOOL DoKeyboardSplit();
// synchronized scrolling
    virtual BOOL DoScroll(CView* pViewFrom, UINT nScrollCode,
        BOOL bDoScroll = TRUE);
    virtual BOOL DoScrollBy(CView* pViewFrom, CSize sizeScroll,
        BOOL bDoScroll = TRUE);
// Implementation **some omitted for brevity
public:
    struct CRowColInfo {
        int nMinSize;           // below that try not to show
        int nIdealSize;         // user set size
        int nCurSize;           // 0 => invisible, -1 => nonexistent
    };
protected:
    CRuntimeClass* m_pDynamicViewClass;
    int m_nMaxRows, m_nMaxCols;
// implementation attributes which control layout of the splitter
    int m_cxSplitter, m_cySplitter;          // size of splitter bar
// space on either side of splitter
    int m_cxBorderShare, m_cyBorderShare;
// amount of space between panes
    int m_cxSplitterGap, m_cySplitterGap;
    int m_cxBorder, m_cyBorder;                // borders in client area
// current state information
    int m_nRows, m_nCols;
    BOOL m_bHasHScroll, m_bHasVScroll;
    CRowColInfo* m_pColInfo;
    CRowColInfo* m_pRowInfo;
// Tracking info - only valid when 'm_bTracking' is set
    BOOL m_bTracking, m_bTracking2;
    CPoint m_ptTrackOffset;
    CRect m_rectLimit;
    CRect m_rectTracker, m_rectTracker2;
    int m_htTrack;

```

```

// implementation routines
    BOOL CreateCommon(CWnd* pParentWnd, SIZE sizeMin, DWORD dwStyle,
                      UINT nID);
    int HitTest(CPoint pt) const;
    void GetInsideRect(CRect& rect) const;
    void GetHitRect(int ht, CRect& rect);
    void TrackRowSize(int y, int row);
    void TrackColumnSize(int x, int col);
    void DrawAllSplitBars(CDC* pDC, int cxInside, int cyInside);
    void SetSplitCursor(int ht);
    CWnd* GetSizingParent();
// starting and stopping tracking
    virtual void StartTracking(int ht);
    virtual void StopTracking(BOOL bAccept);
    afx_msg BOOL OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message);
    afx_msg void OnMouseMove(UINT nFlags, CPoint pt);
    afx_msg void OnPaint();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint pt);
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint pt);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint pt);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
                          CScrollBar* pScrollBar);
    afx_msg void OnVScroll(UINT nSBCode, UINT nPos,
                          CScrollBar* pScrollBar);
    afx_msg BOOL OnNcCreate(LPCREATESTRUCT lpcs);
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnDisplayChange();
DECLARE_MESSAGE_MAP()
};

}

```

You'll notice in this listing that we've left in lots of members in the declaration that are above the “// Implementation” section. Though these members are documented, they are also important to understanding the CSplitterWnd implementation. MFC does a great job of “virtualizing” CSplitterWnd so that you can create CSplitterWnd derivatives and change the default CSplitterWnd behavior. Once you understand more about how CSplitterWnd is implemented, doing so will be even easier.

Here's a rundown of the interesting CSplitterWnd members. Some are documented, some aren't. We'll be seeing most of these again when we start dissecting the CSplitterWnd member functions.

Encapsulated Data Types

- ESplitType—An enumeration that defines the type of splitter to be drawn. Valid types are splitBox, splitBar, splitIntersection, and splitBorder. (Look back to Figure 9-1 to see what each of these splitter window components is.)

- **CRowColumnInfo**—An internal structure that maintains the minimum, ideal, and current size of a row or column. When describing a row, these values refer to the row's height. When describing a column, the values refer to the column's width.

Creation/Layout Data Members

These data members maintain the information needed to create the splitter window panes and the information about the layout of the panes.

- **m_pDynamicViewClass**—A pointer to the CRuntimeClass information for the view (pane) to be dynamically created by CSplitterWnd. This member is specified in Create() for dynamic splitters and in CreateView() for static splitters.
- **m_nMaxRows/m_nMaxCols**—The maximum number of rows/columns as specified in the call to Create() or CreateStatic().
- **m_nRows/m_nCols**—The number of rows and columns that are currently being displayed in the CSplitterWnd.
- **m_bHasHScroll/m_bHasVScroll**—Flags that indicate if a horizontal or vertical scroll bar has been created.
- **m_pColInfo**—An array of CRowInfo, with one element for each column in the CSplitterWnd. This value is fixed in static splitter windows and varies in dynamic splitter windows.
- **m_pRowInfo**—An array of CRowColumnInfo, with one element for each row in the CSplitterWnd. This value is fixed in static splitter windows and varies in dynamic splitter windows.

Cosmetic Data Members

These somewhat “magical” data members (a.k.a. “fudge factors”) are all initialized in the CSplitterWnd constructor. Because they are not documented, we have to assume that the defaults are good for most splitter drawing situations. For fun, you could create a CSplitterWnd derivative and change the values to see the results.

- **m_cxSplitter/m_cySplitter**—The width and height of a splitter box and splitter. This value is 7 for Windows 95 and 4 for Windows NT.
- **m_cxBorderShare/m_cyBorderShare**—These values are 0 if the splitter window is drawing the border (as in Windows 95), or 1 if the border is drawn by Windows (as in Windows NT).
- **m_cxSplitterGap/m_cySplitterGap**—The gap between a splitter box/splitter bar and the scroll bar/border. In Windows NT, this value is 6. In Windows 95, the value is 7.
- **m_cxBorder/m_cyBorder**—The border width of the splitter border. This value is 0 in NT and 2 in Windows 95.

Tracking Data Members

These data members are used for hit testing and tracking.

- **m_bTracking**—If TRUE, the user is dragging one splitter bar.
- **m_bTracking2**—If TRUE, the user is dragging two splitter bars (**m_bTracking** will be true too). When can the user drag two splitter bars? When he or she drags the intersection point.
- **m_ptTrackOffset**—The “pick” size of the hit testing. In other words, the hit testing is not limited to the exact width/height of a split box/splitter bar. **m_ptTrackOffset** gives the user some leeway
- **m_rectLimit**—The size of a pane during tracking, used to determine the height of the tracking splitter bar.
- **m_rectTracker**—The rectangle that is used to draw the splitter bar when tracking.
- **m_rectTracker2**—The rectangle that is used to draw the second splitter bar when tracking (if the user is dragging two bars, via an intersection point).
- **m_htTrack**—Set by the CSplitterWnd hit-tracking mechanism to an enumeration that describes the splitter window component that was “hit.” (More on this guy later.)

General Member Functions

- **CreateCommon()**—Called by both **Create()** and **CreateStatic()** once they have initialized the dynamic- and static-specific members of CSplitterWnd. We’ll look at **CreateCommon()** in detail later.
- **CreateScrollBarCtrl()**—Creates a scroll bar with the specified style and identifier.
- **DoScroll()**—Used to respond to scroll bar messages. **DoScroll()** synchronizes the appropriate panes.
- **DoScrollBy()**—Scrolls the appropriate panes by a specified amount.
- **DoKeyboardSplit()**—Called to programmatically cause the window to split. For example, in Visual C++, you can select Window\Split from the menu to split an editor window.
- **CanActivateNext()**—Called to find out if the next pane can be activated—in other words, if it can be given focus. Called by class **CView**. (More on this later.)
- **ActivateNext()**—Used to activate the next pane. Usually called when a pane is deleted, so that the focus can change before it dies.

Layout Member Functions

- **RecalcLayout()**—This very important member function takes care of all positioning for the splitter window. **RecalcLayout()** is called after a pane is created and also after it is destroyed. We’ll look into **RecalcLayout** further.

- **TrackRowSize()**—Updates the `m_pRowInfo` array information for a specified row. Also determines if there is enough room for the row. If not, `TrackRowSize()` deletes it (in dynamic splitters only).
- **TrackColumnSize()**—Same as above, but for a column.
- **GetSizingParent()**—Retrieves a sizable parent window (only for Windows 95).

Drawing Member Functions

- **DrawAllSplitBars()**—This function “drives” the splitter window-drawing process by making calls to `OnDrawSplitter` for each of the splitter window components to be drawn.
- **OnDrawSplitter()**—`OnDrawSplitter()` draws all the splitter window components. `OnDrawSplitter()` is virtual, so you can modify the appearance of the splitter windows if you want to.
- **OnPaint()**—Called in response to a WM_PAINT message.

Hit Test Member Functions

- **HitTest()**—Takes a point and returns the hit test value for the point.
- **GetInsideRect()**—This member function is similar to `GetClientRect()`, except it takes into account the shared scroll bars.
- **GetHitRect()**—Retrieves the hit rectangle for a specified splitter window component.
- **SetSplitCursor()**—Uses hit testing to determine which cursor to display. If the mouse is over a split box or splitter bar, `SetSplitCursor()` displays the sizing arrows, if the mouse is over an intersection, it displays four-way arrows, and so on.

Tracking Member Functions

- **StartTracking()**—Starts the tracking operation based on a hit test value. Called by `DoKeyboardSplit()` and `OnLButtonDown()`.
- **StopTracking()**—Stops a tracking operation. Called by `OnLButtonDblClk()`, `OnLButtonUp()`, and other functions that want to stop tracking for various reasons.
- **OnInvertTracker()**—Inverts the tracker during rubber-banding.

Message Handlers

- **OnNcCreate()**—`CSplitterWnd` handles the WM_NCCREATE message so that it can remove the WS_EX_CLIENTEDGE extended style bit. `CSplitterWnd` draws the 3-D border itself so that the splitters can appear to be part of the border.

- **OnPaint()**—Draws the splitter window components.
- **OnDisplayChange()**—This is called when the user changes the resolution of the monitor. **OnDisplayChange()** calls **RecalcLayout()** to update the splitter windows for the new setting.
- **OnSize()**—Calls **RecalcLayout()** when the user resizes the splitter window.
- **OnMouseMove()**—Performs hit testing on the splitter window components.
- **OnLButtonDown()**—Also performs hit testing and, if a user clicks on a splitter window component, starts tracking.
- **OnLButtonDblClk()**—Hit tests the double-click, and if the user double-clicks on a split box, it automatically splits the window in half. If the user double-clicks on a splitter bar, the split is removed.
- **OnLButtonUp()**—Stops tracking.
- **OnKeyDown()**—Provides some key sequences that will control splitter window tracking.

There are probably too many members here for you to grasp all at once. At this point, just try to familiarize yourself with them. As we look at specific CSplitterWnd areas, you might want to come back to this section and review what the implementation members are used for.

Now let's look at how splitter windows are initialized. Note that all of CSplitterWnd lives in the WINSPLIT.CPP MFC source file. You may want to open that file with your favorite code editor and follow along.

CSplitterWnd Initialization

The CSplitterWnd constructor is kind of boring. It initializes **m_cxSplitter**, **m_cySplitter**, **m_cxBorderShare**, **m_cyBorderShare**, **m_cxSplitterGap**, **m_cySplitterGap**, **m_cxBorder**, and **m_cyBorder** to the values detailed in the previous section. Let's skip right to the **Create()** and **CreateStatic()** member functions.

Both **Create()** and **CreateStatic()** eventually call **CreateCommon()**, so we will also examine that routine. Finally, we will see how panes are created via the **CreateView()** member function.

Remember from the overview that **Create()** generates dynamic splitter windows. Listing 9-2 contains the pseudocode for **Create()**, from WINSPLIT.CPP.

Listing 9-2. CSplitterWnd::Create(), from WINSPLIT.CPP

```
BOOL CSplitterWnd::Create(CWnd* pParentWnd, int nMaxRows, int nMaxCols,
    SIZE sizeMin, CCreateContext* pContext, DWORD dwStyle, UINT nID)
{
    ASSERT(nMaxRows >= 1 && nMaxRows <= 2);
```

```

ASSERT(nMaxCols >= 1 && nMaxCols <= 2);
ASSERT(nMaxCols > 1 || nMaxRows > 1);           // 1x1 is not permitted
m_nMaxRows = nMaxRows;
m_nMaxCols = nMaxCols;
m_nRows = m_nCols = 1;               // start off as 1x1
if (!CreateCommon(pParentWnd, sizeMin, dwStyle, nID))
    return FALSE;
m_pDynamicViewClass = pContext->m_pNewViewClass;
if (!CreateView(0, 0, m_pDynamicViewClass, sizeMin, pContext))
    DestroyWindow(); // will clean up child windows return FALSE;
m_pColInfo[0].nIdealSize = sizeMin.cx;
m_pRowInfo[0].nIdealSize = sizeMin.cy;
return TRUE;
}

```

Create() starts out with some pretty rigid assertions. These assertions force the CSplitterWnd to be created with either or both nMaxCols and nMaxRows as 2. If the max rows and columns were both 1, there would be no point in having a splitter window. Because dynamic splitters can have only two rows and columns, there are really only three valid ways to call Create()—with 1,2; 2,1; or 2,2 as the max columns and max rows.

If the arguments make it through the assertion gauntlet, Create() stores the nMaxRows/nMaxCols arguments in the corresponding data members. Next, Create() initializes the current number of rows and columns to 1. (m_nRows and m_nCols must always be 1 or 2 in dynamic splitters). A splitter window with one column and one row is just like a window with one view.

After initializing the current and maximum rows and columns, Create() passes the parent window pointer, minimum size, style, and ID to CreateCommon(). We'll look at CreateCommon() soon.

After calling CreateCommon(), Create() stores the CRuntimeClass information for the pane view in m_pDynamicViewClass. Create() then passes the pointer in a call to CreateView(). We'll look at this soon too.

Finally, Create() stores the sizeMin values in the m_pColInfo/m_pRowInfo for the current pane (0,0). This is a little strange because we haven't seen these arrays allocated yet. The allocation must be happening in CreateCommon() or CreateView().

Let's compare Create() to CreateStatic(). Listing 9-3 contains the pseudocode for CreateStatic(), also from WINSPLIT.CPP.

Listing 9-3. CSplitterWnd::CreateStatic(), from WINSPLIT.CPP

```

BOOL CSplitterWnd::CreateStatic(CWnd* pParentWnd, int nRows, int nCols,
    DWORD dwStyle, UINT nID)
{
    ASSERT(nRows >= 1 && nRows <= 16);

```

```

    ASSERT(nCols >= 1 && nCols <= 16);
    ASSERT(nCols > 1 || nRows > 1);
    ASSERT(!(dwStyle & SPLS_DYNAMIC_SPLIT));
    m_nRows = m_nMaxRows = nRows;
    m_nCols = m_nMaxCols = nCols;
    if (!CreateCommon(pParentWnd, CSize(0, 0), dwStyle, nID))
        return FALSE;
    return TRUE;
}

```

In the pseudocode version of CreateStatic(), the initial assertions reveal that static splitter windows have a limit of 16 rows and columns (that's a whopping 256 panes!). The assertions also check that there is at least greater than one row or column (or we wouldn't need a splitter window). The final assertion is explained in the sidebar.

After testing its arguments to death, CreateStatic() sets m_nRows/m_nMaxRows (these are always equal in static splitter windows) to the nRows argument. CreateStatic() does the same for the column counterparts of these data members.

Finally, CreateStatic() calls CreateCommon(), passing along the parent pointer, a size of (0,0), the style, and the integer ID.

Notice that unlike Create(), CreateStatic() does not call CreateView(). It is up to the static splitter window user to create all the views, since the runtime information is not known by CreateStatic().

You've Gotta Have Style—SPLS_DYNAMIC_SPLIT, That Is

In the declaration for Create(), one of the default style bits that is turned on is SPLS_DYNAMIC_SPLIT. This is defined in AFXEXT.H as:

```
#define SPLS_DYNAMIC_SPLIT 0x0001
```

If you ever need to be able to determine if a CSplitterWnd is static or dynamic, this is the only way to tell. The following sample code shows how to determine both static and dynamic splitter windows:

```

if (m_wndSplitter.GetStyle() & SPLS_DYNAMIC_SPLIT){
    // It's dynamic!
    // ...
} else {
    // It's static!
    // ...
}

```

We will see MFC making this check itself in the near future.

Both Create() and CreateStatic() are similar except for the different restrictions they place on their arguments and the fact that Create() calls CreateView() and CreateStatic() does not.

Both functions do call CreateCommon(), so let's take a look and see what other initializations this member function is performing. Listing 9-4 contains the pseudocode for CreateCommon(), from WINSPLIT.CPP.

Listing 9-4. The CSplitterWnd::CreateCommon() pseudocode, from WINSPLIT.CPP

```
BOOL CSplitterWnd::CreateCommon(CWnd* pParentWnd, SIZE sizeMin,
                                DWORD dwStyle, UINT nID)
{
    DWORD dwCreateStyle = dwStyle & ~(WS_HSCROLL|WS_VSCROLL);
    if (afxData.bWin4)
        dwCreateStyle &= ~WS_BORDER;
    if (!AfxDeferRegisterClass(AFX_WNDMDIFRAME_REG))
        return FALSE;
    if (!CreateEx(0, _afxWndMDIFrame, NULL, dwCreateStyle,·
        0, 0, 0, 0, pParentWnd->m_hWnd, (HMENU)nID, NULL))
        return FALSE;           // create invisible
    m_pColInfo = new CRowColInfo[m_nMaxCols];
    for (int col = 0; col < m_nMaxCols; col++){
        m_pColInfo[col].nMinSize = m_pColInfo[col].nIdealSize = sizeMin.cx;
        m_pColInfo[col].nCurSize = -1; // will be set in RecalcLayout
    }
    m_pRowInfo = new CRowColInfo[m_nMaxRows];
    for (int row = 0; row < m_nMaxRows; row++){
        m_pRowInfo[row].nMinSize = m_pRowInfo[row].nIdealSize = sizeMin.cy;
        m_pRowInfo[row].nCurSize = -1; // will be set in RecalcLayout
    }
    SetScrollStyle(dwStyle);
    return TRUE;
}
```

CreateCommon() first creates a temporary style holder, dwCreateStyle, which contains the original dwStyle argument minus the WS_HSCROLL and WS_VSCROLL style bits. CreateCommon() does this because it does not want the Windows window to have scroll bars: the scroll bars need to be managed by the splitter window, not the Windows window.

After adjusting the style flags, CreateCommon() subtracts the WS_BORDER flag if the application is running under Windows 95.

Next, CreateCommon() calls AfxDeferRegisterClass() (see Chapter 2) and then calls CWnd::CreateEx() with no extended styles, _afxWndMDIFrame (the MDI frame window class—no-erase background handling), the handle of the parent from pParentWnd, and nID as the identifier. CreateEx() creates the Windows window, so we're finally doing some creating!

After calling CreateEx(), CreateCommon() allocates and initializes both the m_pColInfo and the m_pRowInfo arrays, using m_nMaxCols/Rows as the size of the

array. CreateCommon() iterates through both the row and the column CRowColInfo arrays and performs these operations:

1. Both nMinSize and nIdealSize are set to the sizeMin argument.
2. The nCurSize is initialized to -1, which indicates that it should be set when the pane's size is initialized (RecalcLayout()).

After initializing both the column and the row CRowColInfo arrays, CreateCommon calls SetScrollStyle() to initialize m_bHasHScroll and m_bHasVScroll and then returns TRUE.

AFX_IDW_PANE_FIRST: What's It All About?

The default ID for both static and dynamic splitter windows is AFX_IDW_PANE_FIRST (or 0xE900, for you hex lovers). MFC also defines a last-pane ID, AFX_IDW_PANE_LAST (0xE9ff), which restricts the number of panes to 256. Having the IDs for all panes fall in this range lets MFC check very easily if a window is a pane just by checking that the ID falls in this range.

When would you ever want to create a splitter window where nID was not AFX_IDW_PANE_FIRST? Great question! What if you had nested splitter windows? Let's say you wanted a static splitter window with two panes: the left pane could be a view and the right pane could be a dynamic splitter window. In this case, you would create the dynamic splitter window with AFX_IDW_PANE_FIRST + 1—indicating that this is not the first pane, but the second (the first pane is the normal view in the parent static splitter window).

There's no end to the flexibility of MFC!

Now that we've found where the actual splitter window is being created (in CreateCommon()), let's look at how the panes are made by CreateView(). Listing 9-5 shows the pseudocode, from WINSPLIT.CPP.

Listing 9-5. CSplitterWnd::CreateView(), from WINSPLIT.CPP

```
BOOL CSplitterWnd::CreateView(int row, int col, CRuntimeClass* pViewClass,
    SIZE sizeInit, CCreateContext* pContext)
{
    m_pColInfo[col].nIdealSize = sizeInit.cx;
    m_pRowInfo[row].nIdealSize = sizeInit.cy;
    BOOL bSendInitialUpdate = FALSE;
    CCreateContext contextT;
    if (pContext == NULL) {
        CView* pOldView = (CView*)GetActivePane();
        if (pOldView != NULL && pOldView->IsKindOf(RUNTIME_CLASS(CView))) {
            contextT.m_pLastView = pOldView;
            contextT.m_pCurrentDoc = pOldView->GetDocument();
            if (contextT.m_pCurrentDoc != NULL)
```

```

        contextT.m_pNewDocTemplate =
            contextT.m_pCurrentDoc->GetDocTemplate();
    }
pContext = &contextT;
bSendInitialUpdate = TRUE;
}
CWnd* pWnd = (CWnd*)pViewClass->CreateObject();
DWORD dwStyle = AFX_WS_DEFAULT_VIEW;
if (afxData.bWin4)
    dwStyle &= ~WS_BORDER;
// Create with the right size (wrong position)
CRect rect(CPoint(0,0), sizeInit);
if (!pWnd->Create(NULL, NULL, dwStyle, rect, this,
    IdFromRowCol(row, col),
    pContext))
    return FALSE;
// send initial notification message
if (bSendInitialUpdate)
    pWnd->SendMessage(WM_INITIALUPDATE);
return TRUE;
}

```

CreateView() initially stores the sizeInit argument in CRowCollInfo's corresponding array indexes. Next, CreateView() sets a local flag, bSendInitialUpdate, to FALSE.

The next block of code is pretty cool. If for some reason the CCreateContext pointer is NULL, CreateView() creates a local CCreateContext() and does its best to initialize each element based on sane values. It calls GetActivePane() to determine the m_pLastView CView pointer. Once CreateView() has the m_pLastView, it can determine the other CCreateContext fields by calling GetDocument() and then CDocument::GetDocTemplate(). After it has found all of these elements, CreateView() points the pContext pointer to them and sets bSendInitialUpdate to TRUE so that they will be initialized correctly later in the function.

Next, CreateView() calls CreateObject() (remember this from Chapter 5?) to create a pane (CWnd derivative) object from the CRuntimeClass information. After creating the pane object, CreateView() sets the styles and the position rectangle, which are passed to a Create() call. The IdFromRowCol() function calculates the ID of the pane based on the row and column. The formula is AFX_IDW_PANE_FIRST + row * 16 + col. So if the pane is row 0 and column 5, the ID is AFX_IDW_PANE_FIRST + 5. If the pane is row 5 and column 0, then the ID is AFX_IDW_PANE_FIRST + 80. This formula guarantees that all 256 possible panes will have a unique identifier in the range from AFX_IDW_PANE_FIRST to AFX_IDW_PANE_LAST (see the sidebar on page 347).

If the call to Create() fails, CreateView() returns FALSE, indicating failure. If Create() succeeds, CreateView() sends a WM_INITIALUPDATE to the pane if the bSendInitialUpdate flag is set. The flag is set only if CreateView() has to find the CCreateContext information on its own. Finally, CreateView() returns TRUE.

Now you've seen all there is to see about initializing and creating a splitter window. After the splitter window and one or more panes have been created with Create() or CreateStatic(), all of the splitter window action takes place in response to window messages and user interactions.

Before looking at how MFC actually draws the neat splitter window gadgets, let's see how MFC manages the different panes.

Pane Management

There are two kinds of pane management in splitter windows. First, we'll look at how panes are dynamically created in dynamic splitter windows. Next, we'll look at how panes are sized and repositioned when the user resizes the frame window that contains the splitter window.

There are two CSplitterWnd members, SplitRow() and SplitColumn(), that are called to dynamically create a pane. Let's randomly pick one, SplitColumn(), and see how it dynamically goes about creating a new column pane.

SplitColumn() takes one argument, the location of the split, and is called when you create a vertical split. The SplitColumn() pseudocode can be found in Listing 9-6, from WINCORE.CPP.

Listing 9-6. The CSplitterWnd::SplitColumn() pseudocode, from WINCORE.CPP

```
BOOL CSplitterWnd::SplitColumn(int cxBefore)
{
    ASSERT(GetStyle() & SPLS_DYNAMIC_SPLIT);
    cxBefore -= m_cxBorder;
    int colNew = m_nCols;
    int cxNew = CanSplitRowCol(&m_pColInfo[colNew-1], cxBefore, m_cxSplitter);
    if (cxNew == -1)
        return FALSE; // too small to split
    // create the scroll bar first (so new views can see that it is there)
    if (m_bHasHScroll &&
        !CreateScrollBarCtrl(SBS_HORZ, AFX_IDW_HSCROLL_FIRST + colNew)) {
        TRACE0("Warning: SplitRow failed to create scroll bar.\n");
        return FALSE;
    }

    m_nCols++; // bump count during view creation
    // create new views to fill the new column (RecalcLayout will position)
    for (int row = 0; row < m_nRows; row++) {
        CSize size(cxNew, m_pRowInfo[row].nCurSize);
        m_pRowInfo[row].nCurSize = cxNew;
        RecalcLayout();
    }
}
```

```

    if (!CreateView(row, colNew, m_pDynamicViewClass, size, NULL))   {
        TRACE0("Warning: SplitColumn failed to create new column.\n");
        // delete anything we partially created 'col' = # columns
        // created
        while (row > 0)
            DeleteView(--row, colNew);
        if (m_bHasHScroll)
            GetDlgItem(AFX_IDW_HSCROLL_FIRST + colNew)->DestroyWindow();
        m_nCols--;           // it didn't work out return FALSE;
    }
}

// new parts created - resize and re-layout
m_pColInfo[colNew-1].nIdealSize = cxBefore;
m_pColInfo[colNew].nIdealSize = cxNew;
RecalcLayout();
return TRUE;
}

```

`SplitColumn()` is not allowed for static splitter windows, so it first asserts that the current splitter window is dynamic by checking the `SPLS_DYNAMIC_SPLIT` flag.

Next, `SplitColumn()` subtracts the size of the border from the column size and sets the local `colNew` variable to the number of columns. `SplitColumn()` then sets the local `cxNew` (the new column width) to the results of calling `CanSplitRowCol()`. `CanSplitRowCol()` calculates the new width of the pane based on the column information, the `cxBefore` value, and the width of a splitter. If the user has created a split smaller than the minimum pane size, `CanSplitRowCol()` will return `-1`, indicating that the pane could not be created. `SplitColumn()` checks for this case and returns `FALSE` if the pane cannot be created. `CanSplitRowCol()` will generate a `TRACE` statement in the debug build. (You may have never noticed it happening, but if you run the Scribble tutorial with splitter windows in debug mode and create a very small split, you will see `CanSplitRowCol()` reject the new pane size and revert the splitter window back to an unsplit window.)

After getting the thumbs-up from `CanSplitRowCol()`, `SplitColumn()` creates the scroll bar for the control if `m_bHasHScroll` is `TRUE`. The `m_bHasHScroll` and `m_bHasVScroll` flags are both set in `SetScrollStyle()` based on the `WS_VSCROLL/WS_HSCROLL` style bits. You may remember that `SetScrollStyle()` was called by `CreateCommon()`. Once the scroll bars are created, `SplitColumn()` increments the number of columns, `m_nCols`, by one.

The next block of code in `SplitColumn()` actually creates the new pane. Notice that this is actually a “for” loop that iterates over the number of columns. `SplitColumn()` does this because if there are two rows and the user creates a new column, then two, not one, new panes need to be created.

As the SplitColumn() “for” loop iterates through the rows creating a new pane for each one, it calls CreateView() with a size based on cxNew for width and the current height of the row for height. If the CreateView() fails, SplitColumn cleans up and returns FALSE.

Once the “for” loop finishes, SplitColumn() updates the CRowCollInfo array elements for the current and previous columns with the post-split column widths. SplitColumn() concludes by calling RecalcLayout() to force a redraw of all of the splitter window panes and gadgets.

Looking through the CSplitterWnd source code, you will notice that at the bottom of almost every member function that has anything to do with pane layout, there is a call to RecalcLayout(). RecalcLayout() literally controls the placement of every splitter window component. Understanding RecalcLayout() is key to understanding how splitter windows work and are updated.

As you can imagine, RecalcLayout() is an extremely long member function, so we cannot reproduce it here. Instead, we will give a general description of the steps RecalcLayout() follows. It’s a good idea to open up the WINSPLIT.CPP file now and look at the code as you read the steps.

1. RecalcLayout() gets the client rectangle by calling ::GetClientRect() and gets the inside rectangle by calling CSplitterWnd::GetInsideRect().
2. RecalcLayout() calculates the row and column layouts in m_pCollInfo/m_pRowInfo by calling LayoutRowCol(). LayoutRowCol() calculates where the rows and columns are located based on the size argument and the size hints stored in the CRowCollInfo argument. RecalcLayout() calls LayoutRowCol() twice: once for columns, with the inside rectangle width, and once for rows, with the inside rectangle height.
3. Next, RecalcLayout() calls ::BeginDeferWindowPos() to set up a Begin/Defer/EndWindowPos call trio. RecalcLayout() uses the number of rows and columns to calculate the number of windows that will be moved. The handle returned by ::BeginDeferWindowPos() is stored in a local AFX_SIZEPARENTPARAMS structure called layout. (See the sidebar for more information on this private MFC structure.)
4. RecalcLayout() next calculates the size of the scroll bars and changes their position (including the size box) by calling a helper, DeferClientPos(), which tweaks the sizes and then calls AfxRepositionWindow(), passing along the pointer to the AFX_SIZEPARENTPARAMS structure. AfxRepositionWindow() lives in WINCORE.CPP. It performs still more checks and finally calls DeferWindowPos using the handle in the AFX_SIZEPARENTPARAMS structure.
5. After repositioning all of the scroll bars, RecalcLayout iterates through the panes and calls DeferClientPos() for each of them with the size information

from the corresponding CRowColumnInfo array element, with some minor adjustments for splitter gaps and other things.

6. Once all of the scroll bars and panes have been repositioned, RecalcLayout() calls ::EndDeferWindowPos() to quickly move all of the windows at once.
7. Finally, RecalcLayout() calls DrawAllSplitBars() with a NULL DC to invalidate the splitter bars and force a redraw in their new positions. The splitter bars will be drawn relative to the new row and column positions.

The AFX_SIZEPARENTPARAMS Private Structure

This private structure is declared in AFXPRIV.H as the following.

```
struct AFX_SIZEPARENTPARAMS
{
    HDWP hDWP;           // handle for DeferWindowPos
    RECT rect;            // parent client rectangle (trim as appropriate)
    SIZE sizeTotal;       // total size on each side as layout proceeds
    BOOL bStretch;         // should stretch to fill all space
};
```

AFX_SIZEPARENTPARAMS is used to send information about a size to a parent window via the MFC window message WM_SIZEPARENT. This structure is important because we will see it used throughout this chapter to help lay out a group of sibling windows.

Now that we've seen how CSplitterWnd goes about laying out the rows and columns, let's see how the splitter bars are drawn.

CSplitterWnd Drawing

The RecalcLayout() member function ended by calling DrawAllSplitBars(), so let's start our drawing exploration there. DrawAllSplitBars() is another lengthy routine, so we will ask you to follow along using your code editor.

Before we look at the details of what DrawAllSplitBars() does, note that this function does not really do any "drawing." The OnDrawSplitter() member function does all of the actual drawing. OnDrawSplitter() takes a DC pointer, an ESplitType enumeration, and a rectangle. We'll look at OnDrawSplitter() after we see how DrawAllSplitBars() drives the function.

First, DrawAllSplitBars() iterates through the columns and draws vertical splitter bars for m_nCols –1. DrawAllSplitBars() calculates the splitter bar dimensions using the width of the splitter, the CRowColumnInfo for the column, and some of the other cosmetic data members.

Next, DrawAllSplitBars() iterates through the rows and performs the identical operations as above, but now using the height of the splitter window and the row information to draw the row splitter bars.

Finally, if the program is running on Windows 95, DrawAllSplitBars() draws a 3-D border around each pane. It does this so that the splitter bars and the border all merge together instead of looking like they were drawn by different programs.

And now the moment you've all been waiting for! Let's look at how CSplitterWnd actually draws all of those fancy splitter boxes and bars. As we mentioned in the DrawAllSplitBars() discussion, OnDrawSplitter() does all of the splitter window drawing.

Listing 9-7 contains the pseudocode for this member function, from WINSPLIT.CPP.

Listing 9-7. The pseudocode for CSplitterWnd::OnDrawSplitter(), from WINSPLIT.CPP

```
void CSplitterWnd::OnDrawSplitter(CDC* pDC, ESplitType nType,
    const CRect& rectArg)
{
    if (pDC == NULL)
        RedrawWindow(rectArg, NULL, RDW_INVALIDATE|RDW_NOCHILDREN);
    return;
    CRect rect = rectArg;
    switch (nType) {
        case splitBorder:
            pDC->Draw3dRect(rect, afxData.clrBtnShadow,
                afxData.clrBtnHilite);
            rect.InflateRect(-CX_BORDER, -CY_BORDER);
            pDC->Draw3dRect(rect, afxData.clrWindowFrame,
                afxData.clrBtnFace);
            rect.InflateRect(-CX_BORDER, -CY_BORDER);
            return;
        case splitIntersection:
            break;
        case splitBox:
            if (afxData.bWin4){
                pDC->Draw3dRect(rect, afxData.clrBtnFace,
                    afxData.clrWindowFrame);
                rect.InflateRect(-CX_BORDER, -CY_BORDER);
                pDC->Draw3dRect(rect, afxData.clrBtnHilite,
                    afxData.clrBtnShadow);
                rect.InflateRect(-CX_BORDER, -CY_BORDER);
                break;
            }
            // fall through...
        case splitBar:
            if (!afxData.bWin4){
                pDC->Draw3dRect(rect, afxData.clrBtnHilite,
                    afxData.clrBtnShadow);
                rect.InflateRect(-CX_BORDER, -CY_BORDER);
            }
            break;
    }
}
```

```

    default:
        ASSERT(FALSE); // unknown splitter type
    }
    pDC->FillSolidRect(rect, afxData.clrBtnFace);
}
}

```

If the CDC pointer passed into OnDrawSplitter() is NULL, this signals that the caller wants to invalidate the rectangle. In this case, OnDrawSplitter() calls CWnd::RedrawWindow() with the rectangle and the RDW_INVALIDATE flag set.

On the other hand, if the CDC pointer is not NULL, OnDrawSplitter switches on the ESplitType argument. Remember from the CSplitterWnd header that this is an enumeration of the various splitter window components (such as splitBorder, splitIntersection, and splitBox).

Before we look at the different cases of the switch statement, notice that after the switch, OnDrawSplitter() calls FillSolidRect() with the local rectangle variable and the button face color. Knowing this, we can deduce that the switch statement at the beginning of FillSolidRect() is adding some “dressing” to the filled rectangle. Let’s look at the switch block and see what OnDrawSplitter() is doing to make the filled rectangles look fancier.

The first case in the switch is splitBorder. The only time that DrawAllSplitBars() calls OnDrawSplitter() with type splitBorder is when the application is running on Windows 95. When OnDrawSplitter() draws the border, it makes two calls to CDC::Draw3dRect(). The first call draws the outside of the border, and the second call draws the inside of the border, using an offset by CX_BORDER, CY_BORDER. This creates a window border effect for the inside of the pane.

The second case in the switch is for splitIntersection, which just breaks out of the switch statement.

The splitBox type is the third case. Let’s use this case to illustrate exactly what these GDI calls are doing and how they result in a cool split box.

Figure 9-2 shows a close-up of a Windows 95 split box and indicates which part of the split box is drawn by each GDI call.



Figure 9-2. Closeup of a Windows 95 split box.

If the application is running on Windows 95, OnDrawSplitter() draws a three-dimensional split box by making two calls to the CDC::Draw3dRect() member function. The first call draws the outside border and the second call draws the inside

border. In Figure 9-2 you can see the results. Notice how Draw3dRect() uses the two argument colors to draw the top and left rectangle sides in one color and the bottom and right rectangle sides in the other color. The result is that Windows 95 “chiseled” 3-D look. If you ever find yourself trying to accomplish that same look, Draw3dRect() is for you. For the Windows 95 case, the splitBox case breaks and falls through to the FillSolidRect() call after the switch block. The FillSolidRect() call completes the split box by filling its center area, as illustrated in Figure 9-2.



Figure 9-3. The Windows NT split box components

If the application is running on Windows NT (or Win32s), the splitBox case falls through to the splitBar case. Figure 9-3 shows the Windows NT split box. Note that in this picture, the black outline is there to provide contrast for the button. This border is not drawn by OnDrawSplitter().

In the splitBar case, OnDrawSplitter() draws some 3-D shading only if it is not running on Windows 95. Remember that this code will be the same for split boxes and split bars on NT and Win32s. The NT splitBox and splitBar cases draw one 3-D rectangle and then deflate the rectangle in preparation for the FillSolidRect() call. Figure 9-3 shows the results of each GDI call on the appearance of the splitter.

Now compare the difference between the Windows 95 split box in Figure 9-2 and the Windows NT split box in Figure 9-3. It's amazing how much "deeper" the Windows 95 split box looks with one extra beveled rectangle.

Now we almost have the complete CSplitterWnd picture. Remember that DrawAllSplitBars() takes care of drawing the splitter bars and the splitter border. This leaves the burning question of what function calls OnDrawSplitter() to draw the split boxes. The answer is OnPaint(). You can find the pseudocode for OnPaint() in Listing 9-8, from WINSPLIT.CPP.

Listing 9-8. The CSplitterWnd::OnPaint() pseudocode, from WINSPLIT.CPP

```
void CSplitterWnd::OnPaint()
{
    CPaintDC dc(this);
    CRect rectClient;
    GetClientRect(&rectClient);
    rectClient.InflateRect(-m_cxBorder, -m_cyBorder);
    CRect rectInside;
    GetInsideRect(rectInside);
    if (m_bHasVScroll && m_nRows < m_nMaxRows)
        OnDrawSplitter(&dc, splitBox, CRect(rectInside.right +
```

```

        afxData.bNotWin4, rectClient.top, rectClient.right,
        rectClient.top + m_cySplitter));
if (m_bHasHScroll && m_nCols < m_nMaxCols)
    OnDrawSplitter(&dc, splitBox,
        CRect(rectClient.left, rectInside.bottom + afxData.bNotWin4,
        rectClient.left + m_cxSplitter, rectClient.bottom));
DrawAllSplitBars(&dc, rectInside.right, rectInside.bottom);
if (!afxData.bWin4) {
    // draw splitter intersections (inside only)
    GetInsideRect(rectInside);
    dc.IntersectClipRect(rectInside);
    CRect rect;
    rect.top = rectInside.top;
    for (int row = 0; row < m_nRows - 1; row++) {
        rect.top += m_pRowInfo[row].nCurSize + m_cyBorderShare;
        rect.bottom = rect.top + m_cySplitter;
        rect.left = rectInside.left;
        for (int col = 0; col < m_nCols - 1; col++) {
            rect.left += m_pColInfo[col].nCurSize + m_cxBorderShare;
            rect.right = rect.left + m_cxSplitter;
            OnDrawSplitter(&dc, splitIntersection, rect);
            rect.left = rect.right + m_cxBorderShare;
        }
        rect.top = rect.bottom + m_cxBorderShare;
    }
}
}

```

OnPaint() prepares to draw by getting the client rectangle and the inside rectangle. Then OnPaint() calls OnDrawSplitter() to draw both the horizontal and the vertical splitter boxes. After drawing the splitter boxes, OnPaint() calls DrawAllSplitBars() to draw the split bars and the split border. Finally, if the application is not running on Windows 95, OnPaint() does lots of work to draw splitter bar intersections.

This block of code raises an interesting question: Why doesn't CSplitterWnd do any special drawing for the Windows 95 intersection? (Remember that OnDrawSplitter() falls through for the splitIntersection case.)

To discover the answer, let's put both the Windows 95 and the Windows NT intersections under the handy MFC internals microscope. The Windows 95 splitter intersection is shown in Figure 9-4, and the Windows NT splitter intersection is shown in Figure 9-5. Each figure shows the commands that draw the various components of the splitter intersection. Can you see why MFC does not need to do any extra 3-D rectangles for Windows 95?

You guessed it! Because MFC draws the splitter window border for Windows 95 applications, the border drawing code does the work of making the split bars look 3-D. Because the bars are hollow (and wider), the splitter intersection is automatically well defined.

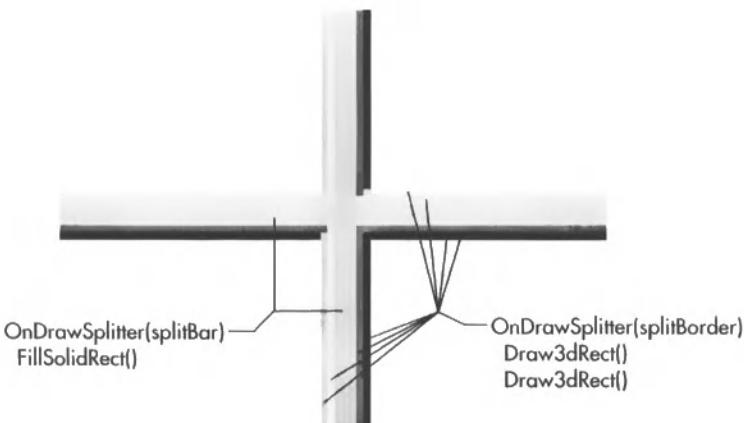


Figure 9-4. A closeup of the Windows 95 splitter intersection

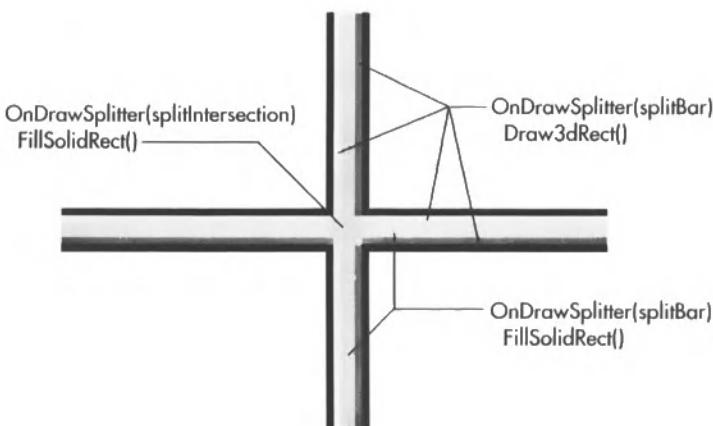


Figure 9-5. The Windows NT splitter intersection under the microscope

On the flip side, the Windows NT application does not draw borders, so it draws the splitter bars with more of a 3-D effect. When the NT application does this, the splitter intersection looks skinny. MFC makes the splitter intersection more defined by drawing an itty-bitty rectangle there (see Figure 9-5).

Now you should understand every aspect of how each splitter window component is drawn. This may seem like MFC trivia to you, but if you ever want or need to change the appearance of the MFC splitter windows, your newly gained knowledge of splitter drawing could be worth your weight in gold!

At this point in the CSplitterWnd adventure, we have two more areas of interest to visit before moving on: hit testing and tracking. Let's first look at what hit testing is and why CSplitterWnd needs it.

Splitter Window Hit Testing

If you've ever written a Windows or MFC control, then the concept of hit testing may be very familiar to you already. If you are a hit-testing neophyte, you need to know that, in a nutshell, hit testing is a technique that lets you programmatically determine if a user-interface action affects one of your control components. In the splitter window case, the splitter window needs to hit-test several user interactions, including these:

- A mouse move, so that the window can change the cursor over splitter bars and boxes.
- A button down, so that the window can drag a bar, create a bar, and so on.
- A button up, so that the window can stop dragging, creating, and so on.

Even if you have written some hit-testing code before, stay tuned: the MFC CSplitterWnd implementation is pretty interesting.

Let's start the CSplitterWnd hit-testing examination by looking at an important internal enumeration found in WINSPLIT.CPP. Listing 9-9 contains the HitTestValue enumeration declaration.

Listing 9-9. The HitTestValue enumeration declaration, from WINSPLIT.CPP

```
// HitTest return values (values and spacing between values is important)
enum HitTestValue
{
    noHit                  = 0,
    vSplitterBox           = 1,
    hSplitterBox           = 2,
    bothSplitterBox        = 3,
    vSplitterBar1          = 101,
    vSplitterBar15         = 115,
    hSplitterBar1          = 201,
    hSplitterBar15         = 215,
    splitterIntersection1 = 301,
    splitterIntersection225 = 525
};
```

HitTestValue enumerates all of the possible hit scenarios. Most of the enumeration value meanings are pretty easy to guess. For example, noHit indicates that none of the splitter window components should be affected, and vSplitterBox indicates that a vertical splitter box is hit.

The first not-so-obvious value in the enumeration is bothSplitterBox. This value is used when the developer programmatically splits a window by calling DoKeyboardSplit().

There are also some puzzling names and values in the enumeration. For example, vSplitterBar1 and vSplitterBar15. Can you guess why these values are named this way and why they have the values 101 and 115?

Remember that the ceiling for rows is 16 in static splitter windows. These values in the enumeration give the hit-testing logic a hit-testing “range” to work with. The same is true for columns. Because there can be a maximum of 256 intersections, HitTestValue has a range from splitterIntersection1 (301) through splitterIntersection225 (525). We’ll see how MFC uses these ranges next when we look at HitTest().

The key to the MFC hit-testing logic is the HitTest() member function. HitTest() takes a point and returns one of the HitTestValue enumerated values. Listing 9-10 shows the HitTest() pseudocode, from WINSPLIT.CPP.

Listing 9-10. The CSplitterWnd::HitTest() pseudocode, from WINSPLIT.CPP

```
int CSplitterWnd::HitTest(CPoint pt) const
{
    CRect rectClient;
    GetClientRect(&rectClient);
    rectClient.InflateRect(-m_cxBorder, -m_cyBorder);
    CRect rectInside;
    GetInsideRect(rectInside);
    if (m_bHasVScroll && m_nRows < m_nMaxRows &&
        CRect(rectInside.right, rectClient.top, rectClient.right,
              rectClient.top + m_cySplitter - afxData.bWin4).PtInRect(pt))
        return vSplitterBox;
    if (m_bHasHScroll && m_nCols < m_nMaxCols && CRect(rectClient.left,
                  rectInside.bottom, rectClient.left + m_cxSplitter - afxData.bWin4,
                  rectClient.bottom).PtInRect(pt))
        return hSplitterBox;
    CRect rect;
    rect = rectClient;
    for (int col = 0; col < m_nCols - 1; col++) {
        rect.left += m_pColInfo[col].nCurSize;
        rect.right = rect.left + m_cxSplitterGap;
        if (rect.PtInRect(pt))
            break;
        rect.left = rect.right;
    }
    rect = rectClient;
    for (int row = 0; row < m_nRows - 1; row++) {
        rect.top += m_pRowInfo[row].nCurSize;
        rect.bottom = rect.top + m_cySplitterGap;
        if (rect.PtInRect(pt))
            break;
        rect.top = rect.bottom;
    }
}
```

```

// row and col set for hit splitter (if not hit will be past end)
if (col != m_nCols - 1) {
    if (row != m_nRows - 1)
        return splitterIntersection1 + row * 15 + col;
    return hSplitterBar1 + col;
}
if (row != m_nRows - 1)
    return vSplitterBar1 + row;
return noHit;
}

```

HitTest() starts out by getting the client rectangle and the inside rectangle. Next, HitTest() creates temporary CRect objects for each splitter window component and calls CRect::PtInRect() with the CPoint argument to determine if the specified point is in the rectangle for the splitter window component.

You may be wondering why HitTest() doesn't just check the window handle of the splitter box and use GetWindowFromPoint() or some other similar technique. Remember that all of the splitter window components are drawn by CSplitterWnd and they all share the same window. HitTest() has to resort to brute-force rectangle calculations because that is the only information available to check against.

After checking for the splitter boxes, HitTest() iterates through the columns looking for a hit. If a hit is found on a row or column, HitTest() breaks out of the row/column loop. After the row and column loops, HitTest() detects a row or column hit from the "for" loops by checking if the current index of the "for" loops (col and row) are the same as m_nRows - 1 or m_nCols - 1. If the index is not at the end of the column/row count, this indicates a hit.

If a column hit is detected, HitTest() also checks for a row hit. If both column and row are hit, then it must be an intersection. If this is the case, HitTest() returns vSplitterIntersection1 + row * 15 + col. This formula gives a unique identifier to each of the 16-by-16 row/column intersection possibilities.

If only a row or only a column is hit, then HitTest() returns hSplitterBar1 for columns or vSplitterBar1 for rows plus the column or row number. If the HitTest() CPoint argument does not fall in any of the CSplitterWnd component rectangles, a value of noHit is returned.

By itself, hit testing is nothing more than a dumb rectangle checker. CSplitterWnd uses it together with tracking to implement the splitter window user interface.

Splitter Window Tracking

Splitter window tracking refers to the "drag-and-drop" user-interface effect implemented by CSplitterWnd. Tracking occurs, for example, when the user clicks on a size bar, drags the splitter bar ghost to the desired location, and drops the splitter bar (by

releasing the mouse button). CSplitterWnd has to perform several operations during tracking. In this section, we'll look at how it implements some of them.

Let's see how tracking works by following the example scenario: the user presses the mouse button on a splitter box, drags the splitter bar ghost, and then releases the button. The operation starts in CSplitterWnd's OnLButtonDown() message handler. Listing 9-11 contains the code, from WINSPLIT.CPP, for OnLButtonDown(). (Remember that the data member flag m_bTracking indicates that the user is currently dragging a splitter bar.)

Listing 9-11. The CSplitterWnd::OnLButtonDown() implementation, from WINSPLIT.CPP

```
void CSplitterWnd::OnLButtonDown(UINT /*nFlags*/, CPoint pt)
{
    if (m_bTracking)
        return;
    StartTracking(HitTest(pt));
}
```

If the user is already tracking, as indicated by m_bTracking, OnLButtonDown returns, because the user cannot have two tracking operations active at the same time. If the user is not already tracking, OnLButtonDown() calls StartTracking() with the return value from HitTest(pt), where pt is the point where the left mouse button was pressed.

StartTracking() is too large to list; basically it uses the HitTest() result to initialize a tracking operation. If HitTest() returns noHit, StartTracking() immediately returns and takes no action. If the point is an intersection hit, StartTracking() prepares to drag two splitter bar ghosts by setting m_bTracking2 to TRUE and initializing m_rectTracker and m_rectTracker2. These rectangles are the starting rectangles for the splitter bar ghosts.

If the split is starting programmatically (bothSplitterBox), StartTracking() does the same operations as it would for an intersection, but instead of starting the ghosts at the current location of an intersection, it starts the ghosts out in the center of the splitter window.

If a point makes it this far in StartTracking(), then either a splitter box or a single splitter bar must have been hit. If this is the case, StartTracking() initializes m_rectTracker with the initial ghost rectangle.

After StartTracking() has initialized based on the hit, it starts the tracking operation by calling SetCapture() to "lock" the mouse input to the splitter window. Next, StartTracking() sets m_bTracking to TRUE and calls OnInvertTracker() for m_rectTracker and m_rectTracker2 if m_bTracking is set.

Finally, StartTracking() stores the hit-test value in m_htTrack and then calls SetSplitCursor() to force a cursor update.

At this point StartTracking is done. It is now up to two other CSplitterWnd message handlers, OnMouseMove() and OnLButtonUp(), to complete the tracking operation. Before examining these two tracking message handlers, let's see what the OnInvertTracker() routine is doing to draw the nifty splitter bar ghosts. Listing 9-12 shows the OnInvertTracker() implementation, from WINSPLIT.CPP.

Listing 9-12. The CSplitterWnd::OnInvertTracker() implementation, from WINSPLIT.CPP

```
void CSplitterWnd::OnInvertTracker(const CRect& rect)
{
    CDC* pDC = GetDC();
    CBrush* pBrush = CDC::GetHalftoneBrush();
    HBRUSH hOldBrush = NULL;
    if (pBrush != NULL)
        hOldBrush = (HBRUSH)SelectObject(pDC->m_hDC, pBrush->m_hObject);
    pDC->PatBlt(rect.left, rect.top, rect.Width(), rect.Height(), PATINVERT);
    if (hOldBrush != NULL)
        SelectObject(pDC->m_hDC, hOldBrush);
    ReleaseDC(pDC);
}
```

OnInvertTracker() is a pretty basic rubber-banding routine. It selects a halftone brush (a brush with a dithered pattern) and then calls CDC::PatBlt() with PATINVERT set. PatBlt() with PATINVERT is an XOR operation that automatically erases the previous PatBlt() to give the impression of rubber-banding. After PatBltting, OnInvertTracker() selects back in the old brush and releases the DC.

Tracking continues after OnLButtonDown() in OnMouseMove(). If m_bTracking is not set, OnMouseMove() calls HitTest() and passes the results to SetSplitCursor() so that the cursor is updated when the user passes the mouse over a splitter window gadget.

If m_bTracking is set (the case we are most interested in), OnMouseMove() moves the tracker to the current mouse location by first erasing the previous splitter bar ghost by calling OnInvertTracker() with the previous rectangle. After the old ghost has been erased, OnMouseMove() calls OnInvertTracker() again to draw the new ghost in the new location based on the coordinates of the mouse move.

Tracking concludes when the user releases the mouse button and CSplitterWnd's OnLButtonUp() message handler is called. OnLButtonUp() calls StopTracking() with TRUE.

StopTracking() releases the capture and erases any ghosts by making a last call to OnInvertTracker(). The next StopTracking() operation depends on the splitter

gadget being tracked as indicated by the stored HitTest() value in m_htTrack. The following list shows the different hit test possibilities and the action performed by StopTracking() for each of them.

- vSplitterBox—Calls SplitRow() to perform a dynamic vertical split.
- hSplitterBox—Calls SplitColumn() to perform a dynamic horizontal split.
- vSplitterBarX—Calls TrackRowSize() to update the internal CRowCollInfo arrays with the new size information, then calls RecalcLayout().
- hSplitterBarX—Calls TrackColumnSize() to update the internal CRowCollInfo arrays with the new size information, then calls RecalcLayout().
- splitterIntersectionX—Calls both TrackRowSize() and TrackColumnSize() to update the internal row and column data for the resized rows and columns. After updating the information, StopTracking() Calls RecalcLayout().
- bothSplitterBox—Calls SplitRow() and SplitColumn() to create both a horizontal and a vertical split in the splitter window.

Once StopTracking() finishes, the splitter window tracking operation is complete.

Now that you understand the various CSplitterWnd details, from initialization to drawing and tracking, let's put it all together by following the CSplitterWnd operations performed when the user creates a dynamic split.

Do Panes Have to Be Views?

Throughout our CSplitterWnd exploits, you may have noticed CSplitterWnd member naming that seemed to imply that panes must be views—for example, CreateView(), m_pNewViewClass, and others.

These members could have been called CreatePane() or m_pNewPaneClass, because any CWnd derivative can be used as a CSplitterWnd pane. Of course, there are some caveats, and it takes more effort to get everything working correctly. For example, the CSplitterWnd synchronized scrolling and activation logic is very view-specific. Also, if you decide to use a CWnd derivative instead of a CView as the pane, you will lose the leverage of the document/view automatic updating that makes splitting the same "view" so easy.

CSplitterWnd Recap

To recap, we will follow an example split through the splitter window functions and tie together each of the splitter window features that we examined in detail. For this example, let's assume that the user creates a vertical split in a previously unsplit dynamic splitter window. After the split, there are two panes (that is, two columns and one row).

Here are the steps that take place, in rough chronological order (see if you can guess the steps before reading them):

1. The CSplitterWnd user initializes the dynamic splitter by calling Create() in a CMDIChildFrame derivative's OnCreateClient() member function.
2. Create() calls CreateCommon(), which in turn calls CWnd::CreateEx() and creates the Windows window for the splitter window.
3. Create() next calls CreateView(), which uses the CRuntimeClass information to make the initial pane.
4. Once everything is created, Windows sends the window a size message. Then RecalcLayout() is called and lays out all of the size bars and the single pane.
5. Next, a paint message is received, which causes OnPaint() to be called. OnPaint() draws all of the splitter gadgets by eventually calling OnDrawSplitter().
6. After the splitter window is created, sized, and painted, nothing happens until the user interacts with the splitter window. In this example, the user presses the button on the bottom left splitter box to create a vertical split.
7. The OnLButtonDown() handler uses hit testing to determine that the user has indeed hit the vertical splitter box. Once OnLButtonDown() detects this, it starts tracking by calling the appropriately named function, StartTracking().
8. As the user moves the mouse around to place the vertical splitter bar, OnMouseMove() is called, which handles the ghost drawing in the splitter window during tracking.
9. When the user drops the splitter bar, the OnLButtonUp() message handler calls StopTracking(), which stops all tracking and then calls SplitColumn().
10. SplitColumn() first checks that there is enough room and then creates both a new scroll bar and a new pane to the right of the splitter bar by calling CreateView(). After the new pane has been created, SplitColumn() calls RecalcLayout().
11. RecalcLayout() updates the position of the original pane (which is cut in half) and the positions of all the other splitter window components. RecalcLayout() invalidates the splitter bars and causes a paint message.
12. OnPaint() is called and all the fresh splitter gadgets are drawn in their correct spot, thanks to the work of RecalcLayout().

We'll bet that on your first trip through the Scribble tutorial, you never guessed that MFC was doing this much work under the hood for you!

For More Information

So far in this chapter there has been a ton of CSplitterWnd information for you to try to digest. If you are a true CSplitterWnd junkie and want to explore even more, follow these suggestions for further CSplitterWnd travels:

- One other interesting aspect of the CSplitterWnd implementation is synchronized scrolling. Investigate synchronized scrolling, starting with DoScroll and DoScrollBy().
- What happens when the user double-clicks on a splitter bar?
- What happens when the user double-clicks on a splitter box?
- What key sequences stop tracking? What key sequences can be used for tracking?
- Other than the ASSERTs in Create(), what are the technical limitations on having more than four panes in a dynamic splitter window?

Now that we've seen how MFC implements its most complex enhanced user-interface class, let's move on to the next set of classes: the MFC control bars.

The MFC CControlBar Architecture

Back in the old days of pre-4.0 MFC, the CControlBar family was much more interesting than it is today. That was before Microsoft rewrote the CStatusBar and CToolBar classes to use the Windows common controls. MFC used to do all of the drawing, pane management, bitmap manipulation, and other jobs for both the tool bar and the status bar; now all of these juicy tasks are handled by the common controls.

But don't panic! We've found some internal CControlBar family facets in the newer implementation of the classes that are equally as interesting. In this section we'll focus mostly on CControlBar and on how it provides the docking and state-storage features for its derivatives.

In fact, after digging deep into the MFC source code, we were surprised to find several (four!) undocumented classes that MFC uses internally to implement these very features. But CControlBar is not just about digging up undocumented classes: there are lots of interesting techniques used internally that we'll be sure to point out.

Enough buildup! Let's start digging.

Step Right Up to the CControlBar

As mentioned, in this section we'll be looking at how CControlBar provides docking ability and toolbar "persistence." Before we look inside CControlBar, let's introduce the CControlBar family and briefly review how to use control bars.

Using CControlBar and Family

CControlBar is a base class that defines common operations for a control bar. A good definition of a control bar is a special window that is usually aligned along one side of a frame window. Some control bars are dockable, meaning that they can be dragged from their position and placed anywhere inside or outside the frame window. Control bars can be considered containers for windows, like buttons and combo boxes, or even as areas for an application to draw on.

In MFC, control bars can also store their positions in the application's INI file or in the registry. When the application restarts, the control bars automatically return to the state they were in when the user last exited the application, thus saving any customizations the user has made.

You may have never heard of CControlBar, because its derived classes are much more popular than the base class itself. The CControlBar derivatives are the following:

- CDialgBar—Used for displaying a group of controls that are created using a dialog template. An example of a CDialgBar is the bar of buttons at the top of the MFC print preview CPreviewView (see Chapter 8). In this case, the CDialgBar acts as a container for Windows windows.
- COleResizerBar—Lets the user resize an OLE object.
- CStatusBar—Contains “frames” that are used to communicate status to the user. The CStatusBar frames are drawn and managed by the Windows common control.
- CToolBar—Contains bitmapped buttons and separators.

The default AppWizard settings will “automagically” add a tool bar and status bar to your application for you. AppWizard adds control bars by following these steps:

1. AppWizard adds data members to your “main” frame window:

```
protected: // control bar embedded members
CStatusBar m_wndStatusBar;
CToolBar   m_wndToolBar;
```

2. In your frame derivative's OnCreate(), AppWizard creates the control bars. Tool bars automatically load the bitmap images from the IDR_MAINFRAME resource that you create using the VC++ resource editor:

```
if (!m_wndToolBar.Create(this) || !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    return -1;      // fail to create
```

Status bar frames, or indicators, are initialized via a static array that AppWizard places at the head of your frame derivative's .CPP file. The static indicator array is then passed to the CStaticBar after calling Create() in the frame derivative's OnCreate().

```

if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/
        sizeof(UINT)))
    return -1;           // fail to create

```

3. Next, AppWizard sets some style bit flags for the tool bar. In the following example, tool tips, fly-by status bar updating, and dynamic sizing are ORed into the original style.

```

m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

```

4. Finally, AppWizard enables docking by adding three small line calls. The first two calls are seemingly duplicates, but the first call is CToolBar::EnableDocking() and the second is CFrameWnd::EnableDocking(). The CBRS_ALIGN_ANY is another control bar style that indicates to MFC that you want the control bar dockable anywhere in the frame window.

```

m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);

```

Wow! Four steps and you've added a docking tool bar and a status bar. In addition to docking, the tool bar supports tool tips and will even display a little explanation about the tool bar button in the status bar (CBRS_FLYBY). If you really want to work up a sweat, you can make the tool bars persistent with two more steps:

5. Save the tool bar state in your CFrame derivative's OnClose() by calling SaveBarState() and passing it a key:

```

CMainFrame::OnClose()
{
    //...
    SaveBarState("ControlBarState");
    //...
}

```

6. Load the tool bar state in OnCreate() (after creating the tool bars, of course). Be sure to use the same key as used in the SaveBarState() call.

```

CMainFrame::OnCreate()
{
    //... create status/toolbar, etc..
    LoadBarState("ControlBarState");
    // ...
}

```

You can even dynamically dock and undock a control bar by calling DockControlBar() and FloatControlBar().

That's a whirlwind tour of the CControlBar family. You can always read more about them in the on-line docs. Let's look at something you can't find in the on-line docs.

Inside CControlBar

If you think about it, docking would be pretty complicated to do on your own. Several hair-pulling issues come to mind:

- How does MFC make the bar pop out of the main window?
- How does MFC decide where to let the bar dock?
- How does MFC implement that neat effect where the ghost of a control bar switches orientation to match its destination?
- How does MFC move windows around to fit a newly docked bar?
- How does MFC draw those neat chiseled edges when the control bar is docked?

To answer these and many other control bar questions, let's see how control bar docking is implemented.

Before getting too deep into docking, let's first take a quick peek at the never-before-seen portions of the CControlBar declaration. CControlBar is declared in AFXEXT.H; Listing 9-13 contains the interesting portions of the declaration. Incidentally, of all MFC classes, the CControlBar implementation is one of the most scattered. The lion's share of the implementation lives in BARCORE.CPP, but there are pieces of CControlBar in BARDOCK.CPP, DOCKSTAT.CPP, and WINFRM.CPP.

Here's a table of the other CControlBar-related MFC source files:

<i>File</i>	<i>Contents</i>
BARDLG.CPP	CDialogBar implementation
BARDOCK.CPP	CControlBar docking helpers
BARSTAT.CPP	CStatusBar implementation
BARTOOL.CPP	CToolBar implementation
DOCKSTAT.CPP	CControlBar docking/persistence helpers
DOCKCONT.CPP	CControlBar dragging helpers

Listing 9-13. The CControlBar implementation details, from AFXEXT.H

```
class CControlBar : public CWnd
{
// Construction **omitted
```

```

// Attributes **omitted
// Operations **omitted
// Overridables **omitted
// Implementation **some omitted
public:
    int m_cxLeftBorder, m_cxRightBorder;
    int m_cyTopBorder, m_cyBottomBorder;
    int m_cxDefaultGap;           // default gap value
    UINT m_nMRUWidth;          // For dynamic resizing.
    int m_nCount;
    void* m_pData;
    enum StateFlags { delayHide = 1, delayShow = 2, tempHide = 4,
                      statusSet = 8 };
    UINT m_nStateFlags;
    // support for docking
    DWORD m_dwStyle;      // creation style (used for layout)
    DWORD m_dwDockStyle;// indicates how bar can be docked
    CFrameWnd* m_pDockSite;
    CDockBar* m_pDockBar;
    CDockContext* m_pDockContext;
    virtual void DoPaint(CDC* pDC);
    void DrawBorders(CDC* pDC, CRect& rect);
    void EraseNonClient();
    void GetBarInfo(CControlBarInfo* pInfo);
    void SetBarInfo(CControlBarInfo* pInfo, CFrameWnd* pFrameWnd);
    void ResetTimer(UINT nEvent, UINT nTime);
// Message handlers
    afx_msg void OnTimer(UINT nIDEvent);
    DECLARE_MESSAGE_MAP()
    friend class CFrameWnd;
    friend class CDockBar;
};


```

Take another glance at Listing 9-13. On your first skim, you may have missed a whopping three undocumented implementation classes: CDockBar, CDockContext, and CControlBarInfo. We'll cover these in detail later (and even two other undocumented classes not seen here!). Before we move on, we want to introduce you to each data member, so you won't be surprised if you see them pop up in the upcoming travels.

Data Members

- m_cxLeftBorder/m_cyLeftBorder—Hard-coded to a value of 6 in the constructor. The various CControlBar derivatives modify these values for different looks.
- m_cyTopBorder/m_cyBottomBorder—Hard-coded to a value of 1 in the constructor. The various CControlBar derivatives modify these values for different looks.

- m_cxDefaultGap—Hard-coded to 2 in the constructor; never modified.
- m_nMRUWidth—Initialized to 32767 in the constructor. This data member is updated to contain the last width of the control bar. The control bar layout logic uses this member in certain situations. The various layout modes are #defined at the top of BARCORE.CPP.
- m_nCount—The number of elements in the control bar. CControlBar derivatives update this member to reflect the number of items in the bar.
- m_pData—A typeless pointer that CControlBar derivatives use to store data. For example, CStatusBar stores an array of AFX_STATUSPANE structures in m_pData. Since this is a void*, the derivatives have to perform lots of casts with the pointer. Kind of ugly.
- m_nStateFlags—Each CControlBar has a state that controls what happens during idle processing, hide/show and
 - delayHide—Delays the hiding of a control bar.
 - delayShow—Delays the showing of a control bar.
 - tempHide—Used by COleResizeBar to indicate that the bar should be temporarily hidden until an OLE operation is complete.
 - statusSet—Indicates whether or not the MFC idle processing should update the status bar with the fly-by information.
- m_dwStyle—Stores the CBRS_XXX style bit flags.
- m_dwDockStyle—Stores some undocumented CBRS_XXX style bit flags that are used by the MFC docking implementation.
- m_pDockSite, m_pDockBar, and m_pDockContext—We'll get to these later.

Member Functions

- DoPaint()—Calls DrawBorders() and is called by OnPaint(). Most derivatives override this call and instead call DrawBorders() in OnNcPaint() to draw in the nonclient window. This simplifies calculations in the client area because everything does not have to be offset by the border width, and so on. (Remember splitter windows?)
- DrawBorders()—Draws the “chiseled” borders around the control bar. CBRS_BORDER, CBRS_BORDER_3D, and other style bit flags affect the appearance of the border. The borders are drawn only when a control bar is docked. CDC::FillSolidRect() is used to draw the borders. Figure 9-6 shows a closeup of some control bar borders. Notice in the figure how DrawBorders() has to know when not to draw a bottom, top, left, or right side based on the docking location of the control bar. As just mentioned, this usually happens in the nonclient area of the control bar window.

- `EraseNonClient()`—`CControlBar` derivatives call this function in their `OnNcPaint()` routine to draw a border in the nonclient area.
- `GetBarInfo()/SetBarInfo()`—We'll get to these later.
- `ResetTimer()`—`CControlBar` has two timers: `ID_TIMER_CHECK` and `ID_TIMER_WAIT`. `ID_TIMER_CHECK` delays the removal of the fly-by information from the status bar. `ID_TIMER_WAIT` inserts a pause before the fly-by information is displayed in case the user is quickly moving over an item and not stopping on it.
- `OnTimer()`—`OnTimer()` handles the two timer situations just described.

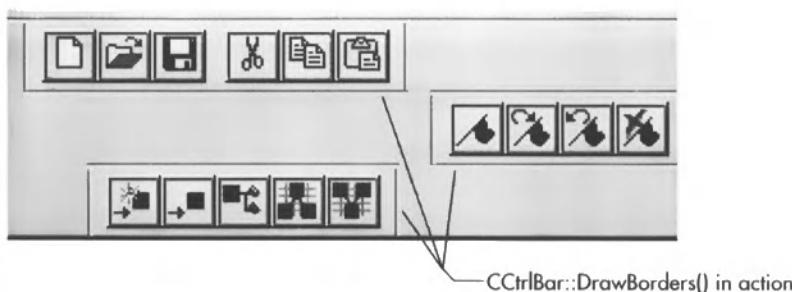


Figure 9-6. A closeup of control bar borders

Now that you've gotten a head start on some `CControlBar` internals, let's look deeply into `CControlBar` and see how docking is implemented by MFC.

CControlBar Docking Implementation

The docking-specific lines added by AppWizard seem like a good place to start, so let's look there first. The first line that AppWizard adds is a call to `CControlBar::EnableDocking()` with `CBRS_ALIGN_ANY`. Listing 9-14 contains the pseudocode for this member function.

Listing 9-14. The `CControlBar::EnableDocking()` pseudocode, from BARDOCK.CPP

```
void CControlBar::EnableDocking(DWORD dwDockStyle)
{
    //**omitted asserts on argument styles
    m_dwDockStyle = dwDockStyle;
    if (m_pDockContext == NULL)
        m_pDockContext = new CDockContext(this);
    if (m_hWndOwner == NULL)
        m_hWndOwner = ::GetParent(m_hWnd);
}
```

EnableDocking() prepares the CControlBar for docking by initializing some data members. First, EnableDocking() stores away the argument docking style in the m_dwDockStyle data member.

Next, EnableDocking() creates a new CDockContext and stores it in the m_pDockContext data member. Finally, EnableDocking() stores the results of ::GetParent() in m_hWndOwner. We realize that you're probably curious about CDockContext, but it's still too early to look at this class. For now, remember that it is initialized in EnableDocking(), which indicates that it will be pretty important later on.

After creating a call to CControlBar::EnableDocking(), AppWizard generated a call to CFrameWnd::EnableDocking(). Let's continue by looking at what the second EnableDocking() routine is doing. Listing 9-15 contains the pseudocode.

Listing 9-15. The CFrameWnd::EnableDocking() pseudocode, from WINFRM2.CPP

```
void CFrameWnd::EnableDocking(DWORD dwDockStyle)
{
    m_pFloatingFrameClass = RUNTIME_CLASS(CMiniDockFrameWnd);
    for (int i = 0; i < 4; i++) {
        if (dwDockBarMap[i][1] & dwDockStyle & CBRS_ALIGN_ANY) {
            CDockBar* pDock = (CDockBar*)GetControlBar(dwDockBarMap[i][0]);
            if (pDock == NULL) {
                pDock = new CDockBar;
                if (!pDock->Create(this, WS_CLIPSIBLINGS|WS_CLIPCHILDREN|
                    WS_CHILD|WS_VISIBLE | dwDockBarMap[i][1],
                    dwDockBarMap[i][0]))
                    AfxThrowResourceException();
            }
        }
    }
}
```

Even to most MFC pros, this is pretty crazy code! The first line is storing the RTCI information for an undocumented MFC class, CMiniDockFrameWnd. We'll look into this later. Next, EnableDocking() iterates from 0 to 3 through a dwDockBarMap and retrieves a CDockBar (another undocumented class!) pointer. If there isn't a pointer, EnableDocking() creates one from the heap and then calls its Create() member function.

You're probably still scratching your head. It helps if you see this 2-D array that is declared in WINFRM2.CPP:

```
const DWORD CFrameWnd::dwDockBarMap[4][2] =
{
    { AFX_IDW_DOCKBAR_TOP,      CBRS_TOP      },
    { AFX_IDW_DOCKBAR_BOTTOM,   CBRS_BOTTOM   },
    { AFX_IDW_DOCKBAR_LEFT,    CBRS_LEFT     },
    { AFX_IDW_DOCKBAR_RIGHT,   CBRS_RIGHT    },
};
```

With this knowledge in hand, you can see that EnableDocking() is iterating through the table and checking the CBRS_XXX (second column) against the dwDockStyle flags. In our example, where dwDockStyle is CBRS_ALIGN_ANY, all four CDockBars are created. If we change the CBRS_ALIGN_ANY argument to the EnableDocking() call to something like CBRS_TOP, then only one CDockBar is created.

EnableDocking() is written so that if all four CDockBars have already been created, they will not be re-created, because GetControlBar() will return the bar and Create() will not be called.

Have you figured out what CDockBar is doing yet? You guessed it: CDockBars are the “landing strips” for dockable toolbars. Figure 9-7 shows all four CDockBars (with IDs AFX_IDW_DOCKBARTOP and so on). We’ll investigate CDockBar in more detail later.

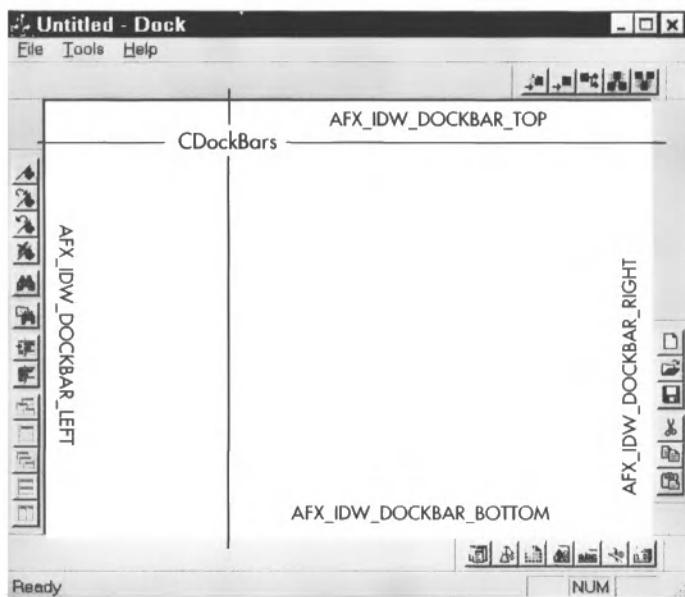


Figure 9-7. CDockBar positions and IDs

One disconcerting aspect of this function is that “new” CDockBar objects are allocated, and then created and seemingly lost because there are no member functions that are set to point to them. Actually, CFrameWnd has an *m_listControlBars* that is a list of CObject pointers. CFrameWnd uses the list to store all control bars for the frame. Control bars are added to the list in CControlBar::OnCreate(). So don’t worry, MFC is keeping track of the CDockBars after all.

Now let's look at the third line added by AppWizard and see what other cool implementation secrets we find. After calling CFrameWnd::EnableDocking(), AppWizard adds a call to CFrameWnd::DockControlBar() that looks like this:

```
DockControlBar(&m_wndToolBar);
```

Listing 9-16 contains the pseudocode for the two CFrameWnd::DockControlBar() member functions, from WINFRM2.CPP

```
void CFrameWnd::DockControlBar(CControlBar* pBar, UINT nDockBarID,
    LPCRECT lpRect)
{
    CDockBar* pDockBar = (nDockBarID == 0) ? NULL :
        (CDockBar*)GetControlBar(nDockBarID);
    DockControlBar(pBar, pDockBar, lpRect);
}

void CFrameWnd::DockControlBar(CControlBar* pBar, CDockBar* pDockBar,
    LPCRECT lpRect)
{
    if (pDockBar == NULL){
        for (int i = 0; i < 4; i++){
            if ((dwDockBarMap[i][1] & CBRS_ALIGN_ANY) == (pBar->m_dwStyle &
                CBRS_ALIGN_ANY)){
                pDockBar = (CDockBar*)GetControlBar(dwDockBarMap[i][0]);
                break;
            }
        }
    }
    pDockBar->DockControlBar(pBar, lpRect);
}
```

The AppWizard generated DockControlBar() call causes the first DockControlBar() override shown in Listing 9-16 to be called. The default argument for the nDockBarID is 0, and lpRect is NULL. The first DockControlBar() detects this case and immediately sets pDockBar to NULL before calling the second DockControlBar().

The second DockControlBar() iterates through the dwDockBarMap table again and calls GetControlBar() to retrieve a CDockBar pointer. Once the pointer is found, DockControlBar calls CDockBar::DockControlBar(), passing in the CControlBar pointer and the rectangle arguments.

Now we're getting somewhere! We've got all of the right catalysts to actually dock a CControlBar. We've got a CDockBar (a landing pad) and a CCtrlBar pointer (a plane) and even a rectangle (a runway number?) that specifies a destination.

CDockBar::DockControlBar() is a pretty long function, so Listing 9-17 contains a fairly abbreviated version. The real McCoy lives in BARDOCK.CPP.

Listing 9-17. An abbreviated CDockBar::DockControlBar(), from BARDOCK.CPP

```
void CDockBar::DockControlBar(CControlBar* pBar, LPCRECT lpRect)
{
    CRect rectBar;
    int nPos = -1;
    pBar->GetWindowRect(&rectBar);
    // **omitted, does some stuff with styles here to 'mark' it as
    // a docked bar and not a floating one.
    if (lpRect != NULL) {
        // insert into appropriate row
        CRect rect(lpRect);
        ScreenToClient(&rect);
        CPoint ptMid(rect.left + rect.Width()/2, rect.top + rect.Height()/2);
        nPos = Insert(pBar, rect, ptMid);
        // position at requested position
        pBar->SetWindowPos(NULL, rect.left, rect.top, rect.Width(),
                           rect.Height(), SWP_NOZORDER|SWP_NOACTIVATE|SWP_NOCOPYBITS);
    }
    else {      // always add on current row, then create new one
        m_arrBars.Add(pBar);
        m_arrBars.Add(NULL);
        // align off the edge initially
        pBar->SetWindowPos(NULL, -afxDATA.cxBorder2, -afxDATA.cyBorder2,
                           0, 0, SWP_NOSIZE|SWP_NOZORDER|SWP_NOACTIVATE|SWP_NOCOPYBITS);
    }
    // attach it to the docking site
    if (pBar->GetParent() != this)
        pBar->SetParent(this);
    if (pBar->m_pDockControl == this)
        pBar->m_pDockControl->RemoveControlBar(pBar, nPos);
    else if (pBar->m_pDockControl != NULL)
        pBar->m_pDockControl->RemoveControlBar(pBar);
    pBar->m_pDockControl = this;
    // remove any place holder for pBar in this dockbar
    RemovePlaceHolder(pBar);
    // get parent frame for recalc layout
    CFrameWnd* pFrameWnd = GetDockingFrame();
    pFrameWnd->DelayRecalcLayout();
}
```

Jackpot! Ever wonder how MFC decides where and how to dock the docking bars that you drag around? DockControlBar() is the culprit. After getting the size of the bar to be docked via GetWindowRect(), DockControlBar() has two different placement scenarios.

In the first scenario, DockControlBar() has been called with a valid lpRect, which specifies a “hint” destination for the dock. DockControlBar takes the rectangle, converts it to client coordinates, calculates a midpoint, and then calls Insert(). After Insert(), DockControlBar() calls SetWindowPos() to position the control bar in the newly calculated position.

In the second scenario, lpRect is NULL; this is the current scenario we are tracing. Remember that in OnCreate(), DockControlBar() was called with the default lpRect, which is NULL. When the lpRect argument is NULL, DockControlBar() appears to add the control bar and NULL to an m_arrBars (meaning an array of bars?) data member. It then calls SetWindowPos() to place the bar exactly on the edge of the window. If you run an AppWizard-generated application, you will notice how the toolbar is always flush left. This is DockControlBar()'s second scenario at work.

Important! Now this brings us to the most important revelation of the section, so pay close attention. After either scenario finishes, the bar to be docked is positioned correctly but is still floating. To zap the floating control bar into the CDockBar, DockControlBar() calls CWnd::SetParent(). SetParent() changes the CControlBar from the child of a floating window to the child of a CDockBar.

If you think about it, SetParent() is really an extremely powerful API. It lets you dramatically change the window hierarchy of an application with very little effort. SetParent() is a great tool for every MFC developer to keep in your mental MFC toolbox.

The check above the SetParent() handles the case where the user is moving a control bar around in the same CDockBar and not actually docking.

After the SetParent() call, DockControlBar() checks m_pDockBar to see if the user is dragging a control bar from one CDockBar to another (such as dragging a bar from the left CDockBar to the right). If this is the case, DockControlBar() calls RemoveControlBar() on the “moved-from” CDockBar to delete the control bar.

To finish up, DockControlBar() calls RemovePlaceHolder(), which cleans up some internal CDockBar structures. Finally, DockControlBar() calls DelayRecalcLayout() on the CFrameWnd pointer returned by GetDockingFrame().

Now you know how MFC implements the docking side of docking toolbars! Before we look at how control bars go from a docking to a floating state, let’s look at the declaration of the already pretty familiar CDockBar. The declaration for CDockBar lives in our favorite MFC header file, AFXPRIV.H, and can also be found in Listing 9-18 for your convenience.

Listing 9-18. The CDockBar declaration, from AFXPRIV.H

```
class CDockBar : public CControlBar
{
    DECLARE_DYNAMIC(CDockControl)
// Construction and Attributes
```

```

public:
    CDockBar(BOOL bFloating = FALSE); //true if attached to
                                      //CMinDockControlWnd
    BOOL Create(CWnd* pParentWnd, DWORD dwStyle, UINT nID);
    BOOL m_bFloating;
// Operations
    void DockControlBar(CControlBar* pBar, LPCRECT lpRect = NULL);
    void ReDockControlBar(CControlBar* pBar, LPCRECT lpRect = NULL);
    BOOL RemoveControlBar(CControlBar*, int nPosExclude = -1);
    void RemovePlaceHolder(CControlBar* pBar);
// Implementation
public:
    virtual CSize CalcFixedLayout(BOOL bStretch, BOOL bHorz);
    virtual void DoPaint(CDC* pDC);
    void GetBarInfo(CControlBarInfo* pInfo);
    void SetBarInfo(CControlBarInfo* pInfo, CFrameWnd* pFrameWnd);
    int FindBar(CControlBar* pBar, int nPosExclude = -1);
    void ShowAll(BOOL bShow);
protected:
    CPtrArray m_arrBars; // each element is a CControlBar
    BOOL m_bLayoutQuery;
    CRect m_rectLayout;
    int Insert(CControlBar* pBar, CRect rect, CPoint ptMid);
// ** Message map entries omitted.
    friend class CMinDockControlWnd;
}

```

Here's a summary of what each CDockBar member does.

Data Members

- **m_bFloating**—This flag indicates whether or not the CDockBar is floating. So far, we've seen only nonfloating CDockBars.
- **m_arrBars**—As we can guess from the name alone, this data member is a CPtrArray in which the pointers are CControlBars.
- **m_bLayoutQuery**—A flag used by the MFC window layout logic.
- **m_rectLayout**—A rectangle used by the MFC window layout logic.

Member Functions

- **Create()**—Initializes the style and then calls CWnd::Create() with the _afxWndControlBar class name.
- **DockControlBar()**—You know what it does!
- **ReDockControlBar()**—This member function quickly moves a floating control bar back to its previous docked state and position and vice versa. This

member is called when the user double-clicks on a CControlBar in an area that does not have a child window or frame. (Try it now in VC++ and you'll see redocking in action.)

- RemoveControlBar()—As the name implies, RemoveControlBar() deletes a control bar from the CDockBar.
- RemovePlaceHolder()—CDockBar keeps the previous location information (a placeholder) in the m_arrBars list in case the user cancels a drag-and-drop. In this situation, CDockBar can zip the bar back to where it belongs. Think of moving it as a two-phase-commit kind of operation.
- CalcFixedLayout()—Lays out all of the control bars in a CDockBar. Does some pretty complex geometry management based on requested rectangles versus actual rectangles and a variety of other factors. CalcFixedLayout() uses the DeferWindowPos() APIs to move all of the control bars quickly. This member is called by the MFC window layout logic.
- GetBarInfo() and SetBarInfo()—More on these later.
- FindBar()—Locates a bar in the internal m_arrBars array.
- ShowAll()—Iterates through the m_arrBars array and shows all of the control bars.
- Insert()—Inserts a new CControlBar into the m_arrBars array. The CControlBars are stored internally in a special order and Insert() maintains the order. CDockBar keeps NULL pointers in the array to denote a new row. Insert() also takes care of maintaining this format.

At this point we've revealed 65% of the CControlBar docking implementation. There are two pieces of information still missing:

1. How does a docking toolbar go the other way and change from a CDockBar child to a floating toolbar?
2. How are the actual dragging and ghost-outline drawing performed, and what class does it?

Let's tackle the first question first. Can you guess what API is doing the work? OK, that was an easy one. Can you guess what class is the CControlbar's parent when it's floating? Stay tuned for the answer, which may surprise you!

CControlBar Floating Implementation

The CFrameWnd::FloatControlBar() member function is called whenever the user wants to float a control bar. Listing 9-19 contains the pseudocode for this function.

Listing 9-19. The CFrameWnd::FloatControlBar() pseudocode, from WINFRM2.CPP

```

void CFrameWnd::FloatControlBar(CControlBar* pBar, CPoint point,
                                 DWORD dwStyle)
{
    // **omitted special case where dragging between floating bars
    // **omitted - some style manipulations.
    CMiDiDockFrameWnd* pDockFrame = CreateFloatingFrame(dwStyle);
    pDockFrame->SetWindowPos(NULL, point.x, point.y, 0, 0,
        SWP_NOSIZE|SWP_NOZORDER|SWP_NOACTIVATE);
    CDockBar* pDockBar =
        (CDockBar*)pDockFrame->GetDlgItem(AFX_IDW_DOCKBAR_FLOAT);
    pDockBar->DockControlBar(pBar);
    pDockFrame->RecalcLayout(TRUE);
    pDockFrame->ShowWindow(SW_SHOWNA);
    pDockFrame->UpdateWindow();
}

```

Notice that `FloatControlBar()` takes three arguments: (1) a `CControlBar` pointer, which is the bar to be floated; (2) a point, which is the upper-left-corner destination for the floating bar; and (3) a style, which specifies some style bits to be applied when the control bar starts floating.

First, `FloatControlBar()` calls `CreateFloatingFrame()`, passing along the style bits to create a `CMiDiDockFrameWnd` (there it is again!). `CreateFloatingFrame()` basically creates a `CMiDiDockFrameWnd` and calls its `Create()` routine. Keep this information on the back burner; we'll return to `CMiDiDockFrameWnd` for a full investigation soon. For now, assume that `CMiDiDockFrameWnd` is a floating frame window (as the name implies).

After creating the `CMiDiDockFrameWnd`, `FloatControlBar()` sets its position to the spot specified by the point argument, using `SetWindowPos()`. Next, a `CDockBar` pointer is retrieved from the `CMiDiDockFrameWnd` and stored in `pDockBar`. After obtaining a `CDockBar` pointer, `FloatControlBar()` makes a familiar call to `DockControlBar()`.

Wow! This is certainly a surprise. Here's the key to understanding what is going on: the `CMiDiDockFrameWnd` has one child, a good old `CDockBar`. In other words, the `CMiDiDockFrameWnd` is a floating landing strip—just like an aircraft carrier.

By using the same class for both the fixed and the floating landing strips, MFC centralizes the docking logic in `CDockBar::DockControlBar()`. In the `FloatControlBar()` case, `SetParent()` will be called, causing the parent to change from a fixed `CDockBar` to the `CMiDiDockFrameWnd`'s floating `CDockBar`.

To conclude, `FloatControlBar()` kicks the MFC Windows layout logic for the `CMiDiDockFrameWnd` by calling `RecalcLayout()`, and it also shows and updates the floating window. From `FloatControlBar()` we learned that the only difference between a fixed and a floating tool bar is really the frame window. In the case of a fixed

CDockBar, the frame is a good old vanilla CFrameWnd. For a floating CDockBar, the frame is a mysterious CMiDiDockFrameWnd.

CMiDiDockFrameWnd is a CMiDiFrameWnd derivative. We'll cover CMiDiFrameWnd later in this chapter. Until then, suffice it to say that CMiDiFrameWnd provides the skinny border around the floating control bars. Let's see what CMiDiDockFrameWnd adds to CMiDiFrameWnd.

CMiDiDockFrameWnd, along with CDockBar, is also declared in our favorite MFC header file, AFXPRIV.H. The declaration is re-created in Listing 9-20 for your convenience.

Listing 9-20. The CMiDiDockFrameWnd declaration, from AFXPRIV.H

```
class CMiDiDockFrameWnd : public CMiDiFrameWnd
{
    DECLARE_DYNCREATE(CMiDiDockFrameWnd)
public:
// Construction
    CMiDiDockFrameWnd();
    virtual BOOL Create(CWnd* pParent, DWORD dwBarStyle);
// Operations
    virtual void RecalcLayout(BOOL bNotify = TRUE);
// Implementation
public:
    CDockBar m_wndDockBar;
    //message map entries omitted -- covered in CMiDiFrameWnd section later
};
```

From Listing 9-20 you can see that CMiDiDockFrameWnd is a very tight CMiDiFrameWnd derivative that adds a custom Create(), a RecalcLayout() member function, and a CDockBar data member called m_wndDockBar.

Let's look quickly at CMiDiDockFrameWnd::Create() to see how the CDockBar is created. Listing 9-21 contains an abbreviated version of Create().

Listing 9-21. An abbreviated CMiDiDockFrameWnd::Create(), from BARDOCK.CPP

```
BOOL CMiDiDockFrameWnd::Create(CWnd* pParent, DWORD dwBarStyle)
{
    /*omitted - recalclayout logic, style settings, etc..
    if (!CMiDiFrameWnd::CreateEx(dwExStyle, NULL, &afxChNil, dwStyle,
        rectDefault, pParent))
        return FALSE;
    CMenus* pSysMenu = GetSystemMenu(FALSE);
    pSysMenu->DeleteMenu(SC_SIZE, MF_BYCOMMAND);
    pSysMenu->DeleteMenu(SC_CLOSE, MF_BYCOMMAND);
    pSysMenu->AppendMenu(MF_STRING|MF_ENABLED, SC_CLOSE, strHide);
    // must initially create with parent frame as parent
```

```

if (!m_wndDockBar.Create(pParent, WS_CHILD | WS_VISIBLE | dwStyle,
    AFX_IDW_DOCKBAR_FLOAT))
    return FALSE;
m_wndDockBar.SetParent(this);
return TRUE;
}

```

Create() first calls CMiFrameWnd::CreateEx() to create a CMiFrameWnd. Next, Create() customizes the CMiFrameWnd by changing around the entries in the menu. Create() then calls CDockBar::Create() to create the internal CDockBar object. Finally, Create() calls SetParent() to change the CDockBar's parent to the CMiDockFrameWnd.

Once you know that the float operation is really just a dock from a CDockBar in a CFrameWnd to a CDockBar in a CMiDockFrameWnd, the floating operation is easy—thanks to the powerful CDockBar helper class.

At this point we've covered about 90% of the control bar docking implementation. Now for the last 10% of the puzzle: how control bars drag.

CControlBar Dragging Implementation

Dragging begins when the user grabs a control bar by moving the cursor over the control bar and pressing the left mouse button. This leads us to CControlBar::OnLButtonDown(), as shown in Listing 9-22.

Listing 9-22. The CControlBar::OnLButtonDown() pseudocode, from BARCORE.CPP

```

void CControlBar::OnLButtonDown(UINT nFlags, CPoint pt)
{
    if (m_pDockControl != NULL && OnToolHitTest(pt, NULL) == -1) {
        ClientToScreen(&pt);
        m_pDockControl->StartDrag(pt);
    }
    else
        CWnd::OnLButtonDown(nFlags, pt);
}

```

OnLButtonDown() calls OnToolHitTest() to verify that the user has clicked on a valid drag start point and not on one of the control bar children or frames. If there is a valid hit, OnLButtonDown() converts the point of the mouse click to screen coordinates and then calls CDockContext::StartDrag(). If the user has not started a drag, OnLButtonDown() is passed to CWnd.

We first encountered CDockContext when we saw one being instantiated and set to m_pDockControl in Listing 9-14 for CControlBar::EnableDocking(). CDockContext handles the dragging operations for CControlBar.

The CDockContext implementation of dragging and tracking is actually pretty similar to the CSplitterWnd tracking implementation. The major difference is that CDockContext draws to a DC that is created from the desktop window and it calls CDC::DrawDragRect() instead of PatBlt() to draw the control bar ghost. CDockContext also takes care of flipping the ghost to show how the toolbar will be oriented when the user stops the dragging operation.

Instead of investigating CDockContext in depth, we'll leave it as a reader exercise. CDockContext is declared in—yep!—AFXPRIV.H. The CDockContext implementation lives in DOCKCONT.CPP. CDockContext remembers the previous location in case the user cancels the drag. The "MRU" data members are used for this purpose.

Now that you completely understand the CControlBar docking implementation, let's investigate how MFC implements the CControlBar persistence.

CControlBar Persistence

You recall from the CControlBar overview that there are two member functions that you need to call in order to save and restore the CControlBar states for your program. Let's look at the save operation, which is initiated with a call to CFrameWnd::SaveBarState(). All of the CControlBar persistence code, including SaveBarState(), lives in DOCKSTAT.CPP. Listing 9-23 contains the pseudocode for SaveBarState().

Listing 9-23. The CFrameWnd::SaveBarState() pseudocode, from DOCKSTAT.CPP

```
void CFrameWnd::SaveBarState(LPCTSTR lpszProfileName) const
{
    CDockState state;
    GetDockState(state);
    state.SaveState(lpszProfileName);
}
```

First, SaveBarState() creates a local CDockState and passes it to GetDockState(). After calling GetDockState() to "load" the state, SaveBarState() writes it to disk by calling CDockState::SaveState().

Well, well, well. It looks as if we have yet another undocumented helper class here! Before we uncover CDockState, let's get a feel for what it's used for by looking at CFrameWnd::GetDockState(). Listing 9-24 contains the pseudocode.

Listing 9-24. CFrameWnd::GetDockState(), from DOCKSTAT.CPP

```
void CFrameWnd::GetDockState(CDockState& state) const
{
    state.Clear();
    POSITION pos = m_listControlBars.GetHeadPosition();
```

```

    while (pos != NULL) {
        CControlBar* pBar = (CControlBar*)m_listControlBars.GetNext(pos);
        CControlBarInfo* pInfo = new CControlBarInfo;
        pBar->GetBarInfo(pInfo);
        state.m_arrBarInfo.Add(pInfo);
    }
}
}

```

From GetDockState() we can actually deduce a great deal about CDockState. First, GetDockState() clears out the state argument to prepare it to receive some fresh control bar implementation. Next, GetDockState() iterates through the CFrameWnd list of control bars.

For each control bar in the frame window, a new CControlBarInfo object is instantiated and passed to CControlBar::GetBarInfo(). After the GetBarInfo() call, the CControlBarInfo object is added to a CDockState internal array of CControlBarInfo.

Egads! Another undocumented CControlBar helper. From the small snippets we've seen, it looks as if CDockState is used to hold the state for multiple control bars. The individual state of a control bar appears to be held in a CControlBarInfo helper. In other words, there will be one CDockState per frame and one CControlBarInfo for each CControlBar in the frame.

CDockState and CControlBarInfo Revealed

Let's look at the declarations for these two new classes and then see what CControlBar::GetBarInfo() and CDockState::SaveState() are doing. Listing 9-25 contains the abbreviated CDockState declaration, from the ultra-hip AFXPRIV.H.

Listing 9-25. The CDockState abbreviated declaration, from AFXPRIV.H

```

class CDockState : public CObject
{
    DECLARE_SERIAL(CDockState)
    CDockState();
public:
// Attributes
    CPtrArray m_arrBarInfo;
public:
// Operations
    void LoadState(LPCTSTR lpszProfileName);
    void SaveState(LPCTSTR lpszProfileName);
    void Clear(); //deletes all the barinfo's
    DWORD GetVersion();
// Scaling Operations
    void ScalePoint(CPoint& pt);
    void ScaleRectPos(CRect& rect);
    CSize GetScreenSize();
}

```

```

void SetScreenSize(CSize& size);
// Implementation
protected:
    BOOL m_bScaling;
    CRect m_rectDevice;
    CRect m_rectClip;
    CSize m_sizeLogical;
    DWORD m_dwVersion;
public:
    ~CDockState();
    virtual void Serialize(CArchive& ar);
};

}

```

Notice in Listing 9-25 that CDockState is a CObject derivative that supports serialization (see Chapter 5 for information on serialization). Also remember that this class contains all of the information needed by a CFrameWnd to store the state of a group of control bars. That state is represented by the data members and manipulated via the member functions. Here's a brief overview of what each member is used for.

First, the CDockState data members:

- m_arrBarInfo—A CPtrArray used to store a CControlBarInfo for each control bar in the frame.
- m_bScaling—A flag that indicates whether the CDockState should scale its values or not. (More on scaling later.)
- m_rectDevice—Stores the size of the display. The display size is determined by calling ::GetSystemMetrics() with SM_CXSCREEN/CYSCREEN.
- m_rectClip—Stores the size of the display minus the size of an icon. The icon size is determined by calling ::GetSystemMetrics() with S_CXICON/CYICON. m_rectClip is used in scaling.
- m_sizeLogical—Stores the logical screen size before scaling is applied to m_rectDevice.
- m_dwVersion—A version used by serialization. In MFC 4.0, the version is 2.

Before we look at the CDockState member functions, a word on scaling. Why does CDockState need to be able to scale? What if a user runs your MFC-based application with the display set to VGA and stores the control bars in certain locations. Then, the next time the user runs your application, he or she first changes the display to 1024×780 . If CDockState then restores all of the control bars to their previous positions, they will be clumped in one corner of the screen, because that is where all the old VGA resolution coordinates live now that the display is running at a much higher resolution.

CDockState gets around this problem by storing the “saved as” resolution. This allows CDockState to determine that the resolution has changed and scale the control bar locations dynamically to fit the new dimensions of the screen. In our example, if the user had a control bar floating in the center of the VGA screen, the CDockState scaling would automatically place that floating bar in the center of the 1024 × 780 screen. Pretty cool feature!

And now for the CDockState member functions:

- LoadState()—Stores the CDockState information in an INI file or the registry.
- SaveState()—Retrieves the CDockState information from an INI file or the registry.
- Clear()—Cleans the CDockState by erasing all of the CControlBarInfo entries in *m_arrBarInfo*.
- GetVersion()—Retrieves the *m_dwVersion* data member.
- ScalePoint()—Scales the specified point using the ratio of the old versus the new resolution. *m_rectClip* is used to make sure that the point is not scaled off of the visible screen.
- ScaleRectPos()—Scales a rectangle using an algorithm similar to ScalePoint().
- GetScreenSize()—Retrieves the size of the screen from *m_rectDevice*.
- SetScreenSize()—When called during CDockState restoration, if the retrieved *m_rectDevice* is not the same as the current screen size, scaling is turned on by toggling *m_bScaling*.
- Serialize()—Serializes a CDockState to an archive. This comes in handy if you want this information to be stored as part of a document instead of in the INI file or the registry. Serialize() serializes out the data members. When the *m_arrBarInfo* array is written out, the CPtrArray calls the ::Serialize() method for each CControlBarInfo. As we’ll see in a second, CControlBarInfo also supports serialization.

Wow! CDockState is a pretty powerful little helper class once you start investigating it. While we’re at it, let’s look at CControlBarInfo too, and see if it has any especially keen features.

Listing 9-26 contains the CControlBarInfo declaration. As you might have guessed, CControlBarInfo’s declaration lives in the hallowed AFXPRIV.H.

Listing 9-26. CDockState’s abbreviated declaration, from AFXPRIV.H

```
class CControlBarInfo
{
public:
```

```

// Implementation
    CControlBarInfo();
// Attributes
    UINT m_nBarID;      // ID of this bar
    BOOL m_bVisible;    // visibility of this bar
    BOOL m_bFloating;   // whether floating or not
    BOOL m_bHorz;       // orientation of floating dockbar
    BOOL m_bDockBar;    // true if a dockbar
    CPoint m_pointPos;  // topleft point of window
    UINT m_nMRUWidth;   // MRUWidth for Dynamic Tool bars
    BOOL m_bDocking;    // TRUE if this bar has a DockContext
    UINT m_uMRUDockID;  // most recent docked dockbar
    CRect m_rectMRUDockPos; // most recent docked position
    DWORD m_dwMRUFloatStyle; // most recent floating orientation
    CPoint m_ptMRUFloatPos; // most recent floating position
    CPtrArray m_arrBarID;  // bar IDs for bars contained within this one
    CControlBar* m_pBar;   // bar which this refers to (transient)
    void Serialize(CArchive& ar, CDockState* pDockState);
    BOOL LoadState(LPCTSTR lpszProfileName, int nIndex,
                   CDockState* pDockState);
    BOOL SaveState(LPCTSTR lpszProfileName, int nIndex);
};

}

```

In Listing 9-26, notice that `CControlBarInfo` is not a `CObject` derivative. Here's what each of the members is used for:

- `m_nBarID`—The ID of the control bar.
- `m_bVisible`—A flag that indicates whether the control bar is visible or not.
- `m_bFloating`—A flag that indicates whether the control bar is docked or floating. (In other words, is its parent a `CFrameWnd` or a `CMiniDockFrameWnd`?)
- `m_bHorz`—A flag that indicates whether the control bar is horizontal or vertical.
- `m_bDockBar`—Is this a `CDockBar`?
- `m_pointPos`—The position of the top left of the window.
- `m_nMRUWidth`—Mirrors the `CControlBar::m_nMRUWidth` data member.
- `m_bDocking`—TRUE if the `CControlBar` has a `CDockContext`.
- `m_uMRUDockID`—The ID of the most recently docked toolbar.
- `m_rectMRUDockPos`—The position of the most recently docked toolbar.
- `m_dwMRUFloatStyle`—The most recently used float style.
- `m_ptMRUFloatPos`—The most recently used float position.
- `m_arrBarID`—The IDs of the bars contained within this one. For example, docked bars can contain multiple `CControlBars`.

- `m_pBar`—A pointer back to the control bar that this `CControlBarInfo` describes.
- `LoadState()`/`SaveState()`—Stores the `CControlBarInfo` data members into an INI or REG file.
- `Serialize()`—Stores the `CControlBarInfo` data members into an archive.

Now that we know what the `CControlBarInfo` data members are, a pretty big architecture question arises. It seems as if `CControlBarInfo` duplicates the `CControlBar` data members. Why not just add `LoadState()`/`SaveState()` and `Serialize()` members to `CControlBar` instead of creating a “mirror” class that will always have to be updated whenever `CControlBar` changes?

How `CControlBar` Persistence Works: Saving

Now that you’ve seen what `CDockState` and `CControlBarInfo` are all about, let’s see how MFC uses them.

Recall from Listing 9-24 that `CFrameWnd::GetDockState()` creates a `CControlBarInfo` and then passes it to `CControlBar::GetBarInfo()`. Let’s see what `GetBarInfo` does with the `CControlBarInfo` object. Listing 9-27 has the `GetBarInfo()` pseudocode.

Listing 9-27. The `CControlBar::GetBarInfo()` pseudocode, from DOCKSTAT.CPP

```
void CControlBar::GetBarInfo(CCControlBarInfo* pInfo)
{
    pInfo->m_nBarID = _AfxGetDlgCtrlID(m_hWnd);
    pInfo->m_pBar = this;
    pInfo->m_bVisible = IsVisible(); // handles delayed showing and hiding
    pInfo->m_nMRUWidth = m_nMRUWidth;
    if (m_pDockControl != NULL) {
        CRect rect;
        GetWindowRect(&rect);
        m_pDockControl->ScreenToClient(&rect);
        pInfo->m_pointPos = rect.TopLeft();
        pInfo->m_bDocking = TRUE;
        pInfo->m_uMRUDockID = m_pDockControl->m_uMRUDockID;
        pInfo->m_rectMRUDockPos = m_pDockControl->m_rectMRUDockPos;
        pInfo->m_dwMRUFloatStyle = m_pDockControl->m_dwMRUFloatStyle;
        pInfo->m_ptMRUFloatPos = m_pDockControl->m_ptMRUFloatPos;
    }
}
```

As you would guess, `GetBarInfo()` basically copies the `CControlBar` state information (the data members) into the `CControlBarInfo` class.

After `GetDockState()` has created a `CControlBarInfo` class and copied the `CControlBar` status into it using `GetBarInfo()` for each of `CControlBar` in the form. After calling

`GetDockState()`, `CFrameWnd::SaveBarState()` (see Listing 9-23) calls `DockState::SaveState()` to save the state.

In Listing 9-23, we saw that after calling `GetDockState()`, `CFrameWnd` calls `CDockState::SaveState()`.

Let's see what `CDockState::SaveState()` does. Listing 9-28 contains it's pseudocode.

Listing 9-28. The abbreviated `CDockState::SaveState()`, from DOCKSTAT.CPP

```
void CDockState::SaveState(LPCTSTR lpszProfileName)
{
    CWinApp* pApp = AfxGetApp();
    int nIndex = 0;
    for (int i = 0; i < m_arrBarInfo.GetSize(); i++) {
        CControlBarInfo* pInfo = (CControlBarInfo*)m_arrBarInfo[i];
        if (pInfo->SaveState(lpszProfileName, nIndex))
            nIndex++;
    }
    TCHAR szSection[256];
    wsprintf(szSection, szSummarySection, lpszProfileName);
    pApp->WriteProfileInt(szSection, szBars, nIndex);
    CSize size = GetScreenSize();
    pApp->WriteProfileInt(szSection, szScreenCX, size.cx);
    pApp->WriteProfileInt(szSection, szScreenCY, size.cy);
}
```

`SaveState()` first iterates through the `m_arrBarInfo` array and calls `CControlBarInfo::SaveState()` for each `CControlBarInfo` structure. After saving each control bar, `SaveState()` stores its own state in the INI or REG file using `WriteProfileInt()`.

`CControlBarInfo::SaveState()` does basically the same thing as `CDockState::SaveState()`. It writes out the `CControlBarInfo` data members using `WriteProfileInt()`.

Let's get a more real-world feel for what's going on by looking at a simple example. Say that we have a vanilla AppWizard application with a status bar and a toolbar. We then add a `SaveBarState("ControlBarState")` to `CMainFrame::OnClose()`. This will cause three bars (the status bar, toolbar, and the toolbar's dockbar) to be saved to the application's INI file.

On exit, the INI file entries will look like this:

[ControlBarState-Summary]

Bars=3

ScreenCX=1280

ScreenCY=1024

[ControlBarState-Bar0]

BarID=59392

```
XPos=-2
YPos=-2
Docking=1
MRUDockID=0
MRUDockLeftPos=218
MRUDockTopPos=257
MRUDockRightPos=206
MRUDockBottomPos=251
MRUFloatStyle=8196
MRUFloatXPos=218
MRUFloatYPos=257
```

[ControlBarState-Bar1]
BarID=59393

[ControlBarState-Bar2]
BarID=59419
Bars=3
Bar#0=65536
Bar#1=59392
Bar#2=65536

CDockState::SaveState() stores the “Summary” entry, which includes the size of the display and the number of control bars. Following the summary is an entry for each of the control bars with the ID, position, and so forth.

You can figure out the type of the control bar by converting it to hex and matching it with the definition from AFXRES.H. For example:

```
Bar0 - 59392 = E800 = AFX_IDW_TOOLBAR = toolbar
Bar1 - 59393 = E801 = AFX_STATUS_BAR = status bar
Bar2 - 59419 = E81B = AFX_DOCKBAR_TOP = toolbar's CDockBar
```

Also notice that the dockbar stores a list of any control bars that it contains. The 65536 indicates that there is no bar in that “slot.”

Pretty neat, eh? If you want to use the registry, call CWinApp::SetRegistryKey() from your application’s InitInstance() and the information will be stored in the registry in a similar format.

How CControlBar Persistence Works: Reading

When the application starts and you call LoadBarState("ControlBarState"), A CDockState is created, which reads the summary information. It then uses the number of bars and iterates through them, creating and loading a CControlBarInfo structure for each one. Because reading is so similar to writing the CControlBar state, we will leave it up to you to explore DOCKSTAT.CPP for the details.

The last aspect of CControlBar left to look at before moving on is how MFC "positions" the control bars.

CControlBar Layout Management

So far we've seen how CControlBars are created, how they are docked, and how they store their state between program executions. The only remaining question is how MFC goes about positioning all of these control bars. If a user resizes the window, what class takes care of resizing the toolbar, the status bar, and so on? When you create a control bar, what class makes room for it?

A thorough examination of the MFC logic at work could take two books this size, so we'll try to show you in general terms how it works, and we'll point you to the more complex areas.

Whenever the state of a frame window changes (for example, a new control bar is added or the window is sized), MFC calls CFrameWnd::RecalcLayout(). Listing 9-29 contains the pseudocode for this function.

Listing 9-29. CFrameWnd::RecalcLayout(), from WINFRM.CPP

```
void CFrameWnd::RecalcLayout(BOOL bNotify)
{
    if (GetStyle() & FWS_SNAPTOBARS) {
        CRect rect(0, 0, 32767, 32767);
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST, reposQuery,
                       &rect, &rect, FALSE);
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST, reposExtra,
                       &m_rectBorder, &rect, TRUE);
        CalcWindowRect(&rect);
        SetWindowPos(NULL, 0, 0, rect.Width(), rect.Height(),
                     SWP_NOACTIVATE|SWP_NOMOVE|SWP_NOZORDER);
    }
    else
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST, reposExtra,
                      &m_rectBorder);
}
```

Basically, RecalcLayout() sets in motion the MFC layout logic by calling CWnd::RepositionBars(). RepositionBars() iterates through the list of window children, finding out the children's current size and their desired size. Then RepositionBars() moves everything around, keeping track of any leftover space and other geometry-management constraints.

After RepositionBars() has calculated the new child window destinations, it calls DeferWindowPos() to move all of the children in one big group instead of individually. RepositionBars() resizes the MDI client window to fit in the remaining room not occupied by control bars.

Of course, this is a vast oversimplification of the process, but it should get you started if you want to explore this topic on your own. Before moving on to CMiniFrameWnd, let's review everything we have learned about CControlBar.

CControlBar Recap

To recap, we uncovered the following highlights about CControlBar:

- CControlBars are drawn by Windows common controls. MFC adds docking, persistence, and layout logic to the common controls.
- CDockBars are the “landing pads” for docking toolbars. They are at work in both docked and floating control bars.
- Floating bars become docked bars by calling SetParent().
- Docked bars become floating bars by calling SetParent(), thus changing the parent window from a CDockBar in a CFrameWnd to a CDockBar in a CMiniDockFrameWnd.
- A CMiniDockFrameWnd is a customized CMiniFrameWnd plus an internal CDockBar that lets the CMiniDockFrameWnd hold control bars and give them floating capabilities—just like aircraft carriers.
- CControlBar dragging is implemented by the undocumented CDockContext helper class.
- The CControlBar state is stored and retrieved using the undocumented CDockState and CControlBarInfo helpers. CDockState is a holder for multiple CControlBarInfo classes. There is a CControlBarInfo class for each CControlBar. These two classes can save and read themselves from an INI file or the registry, thus giving CControlBars the ability to retain their state between application executions.
- MFC control bars are positioned by the fairly complex MFC window layout routines. CWnd::RepositionBars() is the kingpin of this organization.

For More Information

If you like exploring CControlBar as much as we do, here are some good on-ramps to more information:

- Layout logic. Investigate these functions for more information on control bar layout:
 - CDockBar::CalcFixedLayout()
 - CWnd::RepositionBars()
 - CControlBar::OnSizeParent()
- If you are interested in the Macintosh version of MFC, search the BAR*.CPP files for #ifdef _MAC to see the places where the MFC code had to be “tuned” for the Macintosh.
- Check out the CStatusBar implementation in (BARSTAT.CPP):
 - What is AFX_STATUSPANE and where is it stored? (Hint: we told you earlier!)
 - What is SBPF_UPDATE used for?
 - How do the status bar panes get updated?
 - What is CStatusCmdUI?
- Check out the CDialogBar implementation:
 - Where is the document template name stored?
 - How are the child controls and OLE controls initialized?
- Read through the logic in DOCKCONT.CPP to learn more about control bar dragging and tracking.
- Read through the logic in DOCKSTAT.CPP to learn more about control bar persistence.

Before we close out this chapter on the enhanced user-interface features of MFC, we have one last class to cover and a surprise topic, so keep reading.

CMinFrameWnd

In the CControlBar section, we learned a little about the undocumented CMiNiDockFrameWnd class, but we did not see how MFC goes about creating a window with such an itty-bitty frame around it.

Figure 9-8 shows an example of a CMiniFrameWnd. Notice that the title bar is much shorter than that of a normal window. And the menu icon on the left is missing, as well as some of the buttons on the right.



Figure 9-8. A CMiniFrameWnd window

Let's see how MFC goes about getting this "mini" look and feel. From the CMiniFrameWnd declaration in AFXWIN.H, we notice that there are some "unconventional" messages being handled by CMiniFrameWnd:

```
afx_msg BOOL OnNcActivate(BOOL bActive);
afx_msg void OnNcCalcSize(BOOL bCalcValidRects,
    NCCALCSIZE_PARAMS* lpParams);
afx_msg UINT OnNcHitTest(CPoint point); afx_msg void OnNcPaint();
afx_msg void OnNcLButtonDown(UINT nHitTest, CPoint pt );
afx_msg BOOL OnNcCreate(LPCREATESTRUCT lpcs);
```

If you don't know what these messages are used for, don't worry. Even many seasoned Windows programmers don't have occasion to learn about them—which makes them all the more interesting now!

These messages are called "nonclient" messages. The client area of a window is the area inside the borders and title. Windows sends nonclient messages whenever something is happening to a window outside of its client area. For example, when Windows starts drawing your window, it sends a WM_NCPAINT message to draw the title bar and borders. Usually the DefWindowProc() takes care of this message and draws the bar for you. However, Windows is flexible enough to let you snag this message and draw the title yourself. (Ever wonder how Microsoft added the Microsoft logo to the title of Office for Windows 95? Now you know.)

This technique is exactly what CMiniFrameWnd is using to draw its miniaturized window dressings. Of course, in addition to painting the mini-title bar, CMiniFrameWnd has to handle pretty much all of the other nonclient messages, such as activation (the title should look different depending on whether it is activated or not), button clicks in the title, and others.

Besides the nonclient information, the rest of the implementation is a bunch of GDI calls that draw the various window components. We'll leave exploring them to you. You can find all of the code tucked away nicely in WINMINI.CPP. Start with OnNCPaint() and work your way out from there. Happy hunting!

Before we move on to the next chapter, we want to reveal one last MFC user-interface feature: the most recently used file list.

MFC MRU File List Implementation

The MFC most recently used (MRU) file list keeps the user's last couple of files (or documents) easily accessible. The user just has to click on the menu item instead of going through the process of loading the file again.

Before we look at how MFC implements MRUs, let's see how you would use them in an MFC application. AppWizard automatically generates all of the code you need to use MRU files by placing a call to the method `CWinApp::LoadStdProfileSettings()` in your `CWinApp::InitInstance()` function. The argument to `LoadStdProfileSettings()` specifies the maximum number of MRU entries you want. The default is four and the maximum is 16.

Next, all you have to do is tell MFC where you would like MRU file lists placed by inserting a menu item with menu ID `ID_FILE_MRU_FILE1` into your menus using the Visual C++ resource editor.

If your document classes are inherited from `CDocument`, they will automatically save their file names in the MRU list. You can manually manipulate the list via the public methods `CDocument::SetPathName()` and `CWinApp::AddToRecentFileList()`.

How MFC Implements MRU File Lists

The file names for MRU file lists are stored in your application's INI file under the "Recent File List" section name.

When your application calls `CWinApp::LoadStdProfileSettings()`, it instantiates a `CRecentFileList` object and stores it in the `m_pRecentFileList` `CWinApp` data member. Next, `LoadStdProfileSettings()` calls `CRecentFileList::ReadList()`.

Hey, wait a minute. . . . That's another undocumented class! Search all you like, but you will not find a single drop of electronic ink in books on-line dedicated to this class. Let's dig deeper and see what `CRecentFileList` is all about.

The declaration for the class is in Listing 9-30 and is from none other than `AFXPRIV.H`!

Listing 9-30. The `CRecentFileList` declaration, from `AFXPRIV.H`

```
class CRecentFileList
{
// Constructors
public:
    CRecentFileList(UINT nStart, LPCTSTR lpszSection,
                    LPCTSTR lpszEntryFormat, int nSize, int nMaxDispLen =
                    AFX_ABBREV_FILENAME_LEN);
// Attributes
    int GetSize() const;
```

```

CString& operator[](int nIndex);
// Operations
    virtual void Remove(int nIndex);
    virtual void Add(LPCTSTR lpszPathName);
    BOOL GetDisplayName(CString& strName, int nIndex, LPCTSTR lpszCurDir,
        int nCurDir, BOOL bAtLeastName = TRUE) const;
    virtual void UpdateMenu(CCmdUI* pCmdUI);
    virtual void ReadList(); // reads from registry or ini file
    virtual void WriteList(); // writes to registry or ini file
// Implementation
    virtual ~CRecentFileList();
    int m_nSize; // contents of the MRU list
    CString* m_arrNames;
    CString m_strSectionName; // for saving
    CString m_strEntryFormat;
    UINT m_nStart; // for displaying
    int m_nMaxDisplayLength;
    CString m_strOriginal; // original menu item contents
};

};

```

Here's an overview of what each member in this private class does.

Data Members

- **m_nSize**—The number of items in the MRU list.
- **m_arrNames**—A **CString** array that contains the file names.
- **m_strSectionName**—The section name into which to save the MRU information in the INI file. The default is “Recent File List”.
- **m_strEntryFormat**—The string used for the keys of the MRU section. The default is “File%d”, where %d is the position of the file name in the MRU menu.
- **m_nStart**—Specifies the location at which to start placing the menu items. The default is 0.
- **m_nMaxDisplayLength**—The maximum number of MRU items, passed on from **LoadStdProfileSettings()**.
- **m_strOriginal**—The **CString** used to store the original menu string that is being replaced by an MRU menu item.

Member Functions

- **CRecentFileList()**—Initializes all of the data members.
- **GetSize()**—Retrieves the current number of items in the internal MRU file name array.
- **operator[]**—Provides array-like access to the internal MRU file name array.

- Remove()—Removes a file name from the internal MRU list.
- Add()—Adds a file name to the internal MRU list. Checks for duplicates.
- GetDisplayName()—Creates the file name to be displayed, abbreviating the name where necessary. It does this by calling the nifty MFC function AbbreviateName(). This function will add ellipses where needed.
- UpdateMenu()—Called to place an MRU item into the application's menu.
- ReadList()—Reads the MRU list from the application's INI or REG file.
- WriteList()—Stores the MRU list to the application's INI or REG file.

After a CRecentFileList has retrieved the list of menus, MFC calls UpdateMenu() to add the entries to the menu.

When the user wants to load an MRU file, he or she selects the menu, which causes a command message with an ID between ID_FILE_MRUMFILE1 and ID_FILE_MRUMFILE16 to be generated. CWinApp has the following entry in its message map for handling these commands:

```
ON_COMMAND_EX_RANGE(ID_FILE_MRUMFILE1, ID_FILE_MRUMFILE16,
    OnOpenRecentFile)
```

CWinApp::OnOpenRecentFile() uses the ID of the command to look up the file name in the CRecentFileList and then calls OpenDocumentFile() with the file name.

If you are using the document/view architecture, your documents will be automatically added to the MRU menu by CDocument::DoSave(), which makes a call to CWinApp::AddToRecentFileList() with the document name. If you're not using document/views, you can make this call yourself.

Finally, when the application is exiting and CWinApp::ExitInstance() is called, CWinApp calls SaveStdProfileSettings(), which calls m_pRecentFileList->WriteList() to save the CRecentFileList contents to the INI file or the registry.

Conclusion

At this point in the book, you should have a pretty clear understanding of how MFC implements most of the GUI (and some of the non-GUI) classes. Also, if you look in AFXPRIV.H, you'll notice that the classes in there are starting to sound much more familiar to you. If you can name what each of these classes does, then you're well on the way to becoming an MFC internalist.

- CSharedFile
- CPreviewDC
- CPreviewView
- CMiniDockFrameWnd
- CRecentFileList
- CDockState

In the next chapter we will look at the thread classes in MFC and get a better understanding of the different DLL options that MFC provides.

In the chapters after the thread/DLL coverage, we begin our exploration of the MFC OLE classes. Only by understanding these topics can you become a true MFC internalist, so read on.

MFC DLLs and Threads

In this chapter we'll look at two seemingly unrelated topics: MFC DLLs and threads. We'll cover both how MFC works when it is a DLL and how MFC makes it easy for you to write your own extension DLLs that use the class library. This chapter also presents many aspects of the MFC thread implementation and shows how it ties in with the Win32 thread APIs.

Before we dive into each of these topics, let's look at a common concept found in both the MFC DLL and the thread implementations: states.

Understanding States

You may remember that back in Chapter 2 we touched on the AFX_MODULE_STATE structure. In this chapter we revisit that structure in the context of DLLs and threads and also introduce you to some new state structures.

Back in the days of Win16 programming (before Win32 multithreading), having your program's data clobbered was really only a consideration in DLLs. On Win16, if two applications change a variable in a DLL, they both "see" the change.

In Win32 multithreaded applications, data access is more complicated. Win32 provides "process-local" data, thus fixing the Win16 DLL problem. However, threads introduce new data access complexities. Also, Microsoft has to support MFC on Win32s, which does not have process-local data, so they have to worry about those annoying Win16 DLL issues.

To solve these problems, MFC implements separate state objects that contain the state needed by different MFC constructs.

MFC States Explained

There are three types of MFC state information: module state, process state, and thread state. In Win32, a module is executable code that operates independently of the rest of an application. Examples are DLLs and OLE controls. Even MFC is a module when used as a DLL. A process is the same as a Win32 process.

Process states and thread states are very similar: they would have traditionally been global variables, but they have to be specific to a given process or thread for proper Win32s or multithreading support.

A module state is unique because it can contain either a truly global state or a state that is process local or thread local, and it can be switched very quickly. For example, if three programs are using the MFC DLL, there will be a module state for each application that contains the MFC DLL state for each of the users.

Think of the MFC states as localizing data to different chunks of your program. For example, the process state contains data that is local to the process and a module. The module state contains data that is local to a module, and the thread state contains information that is local to a thread.

MFC states can be a pretty confusing concept, so let's stop dealing in abstract terms and look at some code! The next three subsections analyze each of the three key MFC states, from most general (process) to most specific (thread).

Note that when we look at the different states, you will see `#ifdef _AFXDLL` sections of code. `_AFXDLL` is on when MFC is built as a DLL and off when it is built statically. MFC needs different information in each of these scenarios because of the different requirements placed on it in a static versus shared environment. We'll cover these in detail when we look more closely at DLLs. For now, you may want to make a mental note of the differences.

The MFC Process State

What kind of information would need to be process local? Listing 10-1 shows the `AFX_MODULE_PROCESS_STATE` definition, from `AFXSTAT_.H`.

Listing 10-1. AFX_MODULE_PROCESS_STATE, from AFXSTAT_.H

```
class AFX_MODULE_PROCESS_STATE : public CNoTrackObject
{
public:
    AFX_MODULE_PROCESS_STATE();
    virtual ~AFX_MODULE_PROCESS_STATE();
    void (PASCAL *m_pfnFilterToolTipMessage)(MSG*, CWnd*);
#ifndef _AFXDLL
    // CDynLinkLibrary objects (for resource chain)
    CTypedSimpleList<CDynLinkLibrary*> m_libraryList;

```

```

// special case for MFCxxLOC.DLL (localized MFC resources)
HINSTANCE m_appLangDLL;
#endif
    // OLE control container manager
    COccManager* m_pOccManager;
    // locked OLE controls
    CTypedSimpleList<COleControlLock*> m_lockList;
};

}

```

Aside from the constructor and destructor, AFX_MODULE_PROCESS_STATE contains the following:

- m_pfnFilterToolTipMessage—A function pointer for a function that filters tool tip messages.
- m_libraryList—A linked list of “attached” MFC extension DLLs (more on this later).
- m_appLangDLL—An instance handle for the current localized resources.
- m_pOccManager—A pointer to an OLE control manager object (more on this in Chapter 15).
- m_lockList—A linked list of locked OLE controls (more on this in Chapter 15).

So at the process state level, there’s really not too much of interest to us MFC developers. But it’s still good to be aware that this beast exists, in case you ever run across it. We’ll focus on the module and thread states in this chapter because they are more relevant to your day-to-day programming. After looking at the contents of each state, we will examine how they are all related. For fun, you may want to count the number of times we use the word *state*.

Inside the MFC Module State

What kinds of data need to be module local? Each DLL has a resource, so there’s bound to be some way of keeping all of the resources straight. Also, when a DLL is initialized, it is usually passed an hInstance.

Listing 10-2 shows the declaration of AFX_MODULE_STATE so you can see firsthand what information MFC needs as module local.

Listing 10-2. The AFX_MODULE_STATE definition, from AFXSTAT_.H

```

class AFX_MODULE_STATE : public CNoTrackObject
{
public:
#ifndef _AFXDLL
    AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion);

```

```

AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion,
    BOOL bSystem);
#else
    AFX_MODULE_STATE(BOOL bDLL);
#endif
CWinApp* m_pCurrentWinApp;
HINSTANCE m_hCurrentInstanceHandle;
HINSTANCE m_hCurrentResourceHandle;
LPCTSTR m_lpszCurrentAppName;
BYTE m_bDLL;
BYTE m_bSystem;
BYTE m_bReserved[2];
short m_fRegisteredClasses;
#ifdef _AFXDLL
    CRuntimeClass* m_pClassInit;
#endif
CTypedSimpleList<CRuntimeClass*> m_classList;
#ifdef _AFXDLL
    ColeObjectFactory* m_pFactoryInit;
#endif
CTypedSimpleList<ColeObjectFactory*> m_factoryList;
long m_nObjectCount;
BOOL m_bUserCtrl;
TCHAR m_szUnregisterList[4096];
#ifdef _AFXDLL
    WNDPROC m_pfnAfxWndProc;
    DWORD m_dwVersion;
#endif
PROCESS_LOCAL(AFX_MODULE_PROCESS_STATE, m_process)
THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)
};

```

MFC keeps the following data in the AFX_MODULE_STATE:

- m_pCurrentWinApp—A pointer to the CWinApp object for this module.
- m_hCurrentInstanceHandle—The instance handle for the module.
- m_hCurrentResourceHandle—The instance handle for resources. This is different if you want to have a language-local resource.
- m_lpszCurrentAppName—The current application name.
- m_bDLL—A flag indicating if this module is a DLL.
- m_bSystem—A flag indicating if this module is a system module.
- m_fRegisteredClasses—Bitflags for the module's delay-registered classes.
- m_pClassInit—A pointer to the CRuntimeClass information for the first class (usually the head of m_classList).

- `m_classList`—A list of CRuntimeClass information for objects in the module.
- `m_pFactoryInit`—A pointer to the first COleObjectFactory object (usually the head of `m_factoryList`).
- `m_factoryList`—A list of COleObjectFactory objects for the module.
- `m_nObjectCount`—The number of locked OLE objects.
- `m_bUserCtrl`—Set to TRUE if the user has control, FALSE if locked out.
- `m_szUnregisterList`—A list of registered classes to be unregistered.
- `m_pfnAfxWndProc`—A pointer to the module-specific AfxWndProc.
- `m_dwVersion`—The version of MFC that this module linked against.
- `m_process`—Process state information.
- `m_thread`—Thread state information.

Notice that the `m_process` and `m_thread` declarations are wrapped by the `PROCESS_LOCAL` and `THREAD_LOCAL` macros. These macros wrap the data with classes that place the data into different areas. In the `THREAD_LOCAL` case, the data is wrapped by a class that uses the Win32 Thread Local Storage (TLS) APIs, such as `TlsSetValue()`, `TlsAlloc()`, `TlsFree()`, and `TlsGetValue()`. MFC maintains an internal array of TLS values in the `CThreadSlotData` class. We leave it as a reader exercise to explore this class and the MFC implementation use of TLS, as defined in the `AFXTLS_.H` file and implemented in `AFXTLS.CPP`.

`PROCESS_LOCAL` uses TLS to protect data in DLLs running on Win32s, which does not provide process-local data at the operating-system level.

Let's look at the thread state contents and then start piecing together the MFC state puzzle.

The MFC Thread State

What kind of information does a thread need? Remember that a thread is a thread of execution (think of threads as big code worms eating their way through applications). On a thread's journeys through MFC, it will need to keep track of a variety of information.

Remember that a thread can be interrupted at any time, so MFC has to be very careful to put all of the data in the thread state to recover from an interrupted MFC call.

Listing 10-3 contains the declaration for the `MFC _AFX_THREAD_STATE` class.

Listing 10-3. The `_AFX_THREAD_STATE` declaration, from `AFXSTAT_.H`

```
class _AFX_THREAD_STATE : public CNoTrackObject
{
public:
    _AFX_THREAD_STATE();
```

```

virtual ~_AFX_THREAD_STATE();
AFX_MODULE_STATE* m_pModuleState;
AFX_MODULE_STATE* m_pPrevModuleState;
void* m_pSafetyPoolBuffer;      // current buffer
AFX_EXCEPTION_CONTEXT m_exceptionContext;
CWnd* m_pWndInit;
CWnd* m_pAlternateWndInit;      // special case commdlg hooking
DWORD m_dwPropStyle;
DWORD m_dwPropExStyle;
HWND m_hWndInit;
BOOL m_bDlgCreate;
HHOOK m_hHookOldSendMsg;
HHOOK m_hHookOldCbtFilter;
HHOOK m_hHookOldMsgFilter;
MSG m_lastSentMsg;             // see CWnd::WindowProc
HWND m_hTrackingWindow;        // see CWnd::TrackPopupMenu
HMENU m_hTrackingMenu;
TCHAR m_szTempClassName[96];    // see AfxRegisterWndClass
HWND m_hLockoutNotifyWindow;   // see CWnd::OnCommand
BOOL m_bInMsgFilter;
CView* m_pRoutingView;         // see CCmdTarget::GetRoutingView
CFrameWnd* m_pRoutingFrame;    // see CCmdTarget::GetRoutingFrame
BOOL m_bWaitForDataSource;
CToolTipCtrl* m_pToolTip;
CWnd* m_pLastHit;              // last window to own tooltip
int m_nLastHit;                // last hit-test code
TOOLINFO m_lastInfo;           // last TOOLINFO structure
int m_nLastStatus;              // last flyby status message
CCtrlBar* m_pLastStatus;        // last flyby status control bar
};


```

_AFX_THREAD_STATE contains the following members:

- **m_pModuleState**—A pointer to the current module state.
- **m_pPrevModuleState**—A pointer to the previous module state.
- **m_pSafetyPoolBuffer**—A pointer to a safety buffer, allowing for robust temporary object allocation.
- **m_exceptionContext**—The current exception context.
- **m_pWndInit**—A pointer to the most recently hooked window.
- **m_pAlternateWndInit**—A pointer to the most recently hooked common dialog window.
- **m_dwPropStyle**—A property page style.
- **m_dwPropExStyle**—A property page extended style.
- **m_hWndInit**—The matching handle for **m_pWndInit**.

- `m_bDlgCreate`—Indicates that a dialog is being created. MFC performs different background coloring for dialogs from that of normal windows.
- `m_hHookOldSendMsg`—A handle to the previous handle returned from `::SetWindowsHookEx(WH_CALLWNDPROC)`.
- `m_hHookOldCbtFilter`—A handle to the previous handle returned from `::SetWindowsHookEx(WH_CBT)`.
- `m_hHookOldMsgFilter`—A handle to the previous handle returned from `::SetWindowsHookEx(WH_MSGFILTER)`.
- `m_lastSentMsg`—The last message sent.
- `m_hTrackingWindow`—The handle of the current tracking window.
- `m_hTrackingMenu`—The handle of the current tracking menu.
- `m_szTempClassName`—A buffer used in `AfxRegisterWndClass()`.
- `m_hLockoutNotifyWindow`—The handle of a locked-out (no OLE controls) window, if one exists.
- `m_bInMsgFilter`—A flag that indicates that the thread is in a message filter.
- `m_pRoutingView`—A view saves itself here before routing messages to a document.
- `m_bWaitForDataSource`—Indicates that ODBC is waiting for data.
- `m_pToolTip`—A pointer to the current `CToolTipCtrl`.
- `m_pLastHit`—A pointer to the last window to own the tool tip control.
- `m_nLastHit`—The last hit test code (for tool tip hit testing).
- `m_lastInfo`—The last tool tip `TOOLINFO` structure.
- `m_nLastStatus`—The last fly-by status code.
- `m_pLastStatus`—A pointer to the last fly-by status control bar.

How the MFC States Are Related

Now that you've gotten a taste of the types of information stored in the MFC state structures, let's see how they all relate, starting with the thread and working our way back up to the process.

When MFC needs to get to the current `_AFX_THREAD_STATE`, it calls `AfxGetThreadState()`, which is implemented as follows:

```
_AFX_THREAD_STATE* AFXAPI AfxGetThreadState()
{
    return _afxThreadState.GetData();
}
```

Earlier in AFXSTATE.CPP is a global declaration:

```
THREAD_LOCAL(_AFX_THREAD_STATE, _afxThreadState)
```

This line creates an `_AFX_THREAD_STATE` class in every thread's TLS called `_afxThreadState`, which you can conveniently access by calling `AfxGetThreadState()`.

Recall from Listing 10-3 that `_AFX_THREAD_STATE` maintains a pointer to the current module state called `m_pModuleState`. In MFC, the `AFX_MODULE_STATE` is retrieved by calling `AfxGetModuleState()`, which is implemented as follows:

```
AFX_MODULE_STATE* AFXAPI AfxGetModuleState()
{
    _AFX_THREAD_STATE* pState = _afxThreadState;
    AFX_MODULE_STATE* pResult;
    if (pState->m_pModuleState != NULL)
        pResult = pState->m_pModuleState;
    else
        pResult = AfxGetAppState(); return pResult;
}
```

In most cases `AfxGetModuleState()` will get the `_afxThreadState.m_pModuleState` `AFX_MODULE_STATE`. If this is `NULL`, there is a global process-local module state that is defined as follows:

```
PROCESS_LOCAL(_AFX_BASE_MODULE_STATE, _afxBaseModuleState)
```

`AfxGetAppState()` is implemented as follows:

```
AFX_MODULE_STATE* AFXAPI AfxGetAppState()
{
    return _afxBaseModuleState.GetData();
}
```

The `GetData()` call does the magic of retrieving the information out of TLS by calling `::TlsGetValue()` after looking up the data's slot in an internal MFC list. Figure 10-1 shows how modules and threads are related.

When MFC is a DLL, it changes the module state by calling `AfxSetModuleState()`, which is implemented as follows:

```
AFX_MODULE_STATE* AFXAPI AfxSetModuleState(AFX_MODULE_STATE* pNewState)
{
    _AFX_THREAD_STATE* pState = _afxThreadState;
    AFX_MODULE_STATE* pPrevState = pState->m_pModuleState;
```

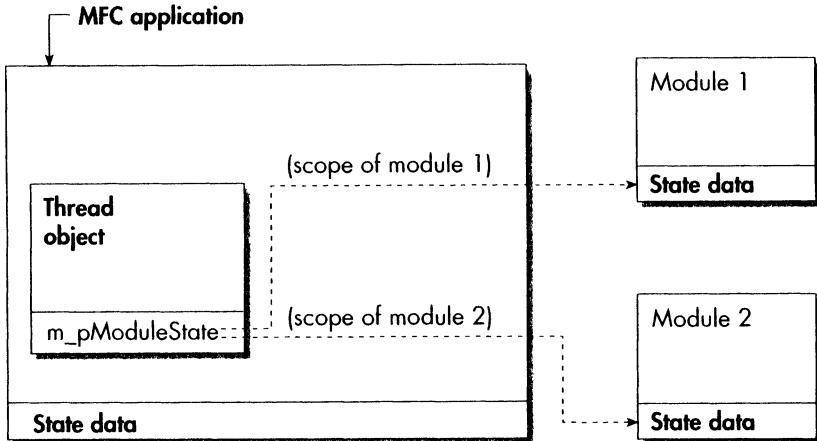


Figure 10-1. The relationship between threads and modules

```

    pState->m_pModuleState = pNewState;
    return pPrevState;
}

```

Now that we have introduced you to the concept of MFC states, we can start our coverage of the “real” DLL and thread issues. We’ll start with DLLs and then look at MFC threads. Along the way, you will be seeing much more of these state structures, so make sure you have a pretty good grasp of what they are up to.

MFC DLLs

The MFC DLL support has been gradually improving since MFC 2.0. In the old days, there used to be two kinds of DLL schemes: USRDLLs and AFXDLLs (named for the definitions that they use).

USRDLLs can be statically “attached” to a DLL. This means that you can use MFC in a DLL without requiring the application using your DLL to be written in MFC. The problem with USRDLLs is that they result in gigantic DLLs, because you are grafting MFC onto your DLL. If you have three or four of these DLLs, it gets pretty crazy. USRDLLs also fall short of the main goal of DLLs: to reduce code duplication.

AFXDLLs are DLLs that use MFC while MFC is in a DLL or statically linked to an application. AFXDLLs used to be limited to just MFC applications (thus USRDLLs were born).

In today's MFC 4.x, USRDLLs are being phased out and called regular DLLs. In other words, they are vanilla Win32 DLLs that have an export list. They can be statically linked, which is the old USRDLL way of doing things, or they can even be dynamically linked and shared (this is new in 4.0). Usually this configuration is used if a DLL writer wants to export the functions of the DLL to a non-MFC application but still use MFC in the DLL. AFXDLLs also have a new and friendlier name: extension DLLs.

In this section we'll focus on the MFC extension DLL implementation. We'll concentrate on the differences between the DLL build of MFC and the non-DLL build of MFC. First, let's discuss resources.

DLL Resource Issues

You may have noticed in the MFC source code that whenever a resource is loaded, instead of directly loading the resource, Microsoft calls `AfxFindResourceHandle()`. This helper function actually takes care of searching through a list of loaded extension DLLs looking for the resource. Before we look at pseudocode for `AfxFindResourceHandle()`, let's first see how MFC maintains a list of extension DLLs and why.

Actually, three types of information live in DLLs and have to be "chained" together for MFC to be able to find the information at run time: resources, static CRuntimeClass (CObject derivatives—see Chapter 5) pointers, and OLE object factories.

Let's look at how MFC goes about storing, searching for, and retrieving this information. We'll focus on the topic of resources, but the CRuntimeClass information and OLE object factory information is stored and chained in the same manner.

The trail begins in the `DllMain()` function (called whenever a DLL is loaded) for the DLL version of MFC, where we find the cryptic lines illustrated by Listing 10-4.

Listing 10-4. A snippet of code from DllMain()

```
AfxInitExtensionModule(coreDLL, hInstance)
CDynLinkLibrary* pDLL = new CDynLinkLibrary(coreDLL, TRUE);
```

In fact, if you ever write an MFC extension DLL, you will have to add these same cryptic lines to your `DllMain()`. The argument to both `AfxInitExtensionModule()` and the `CDynLinkLibrary` constructor is an `AFX_EXTENSION_MODULE`. This is a small structure defined in `AFXDLL_.H` that contains information needed by each MFC extension DLL. `AFX_EXTENSION_DLL` is declared as follows:

```
struct AFX_EXTENSION_MODULE
{
    BOOL bInitialized;
    HMODULE hModule;
```

```

HMODULE hResource;
CRuntimeClass* pFirstSharedClass;
ColeObjectFactory* pFirstSharedFactory;
};

}

```

Let's look at AfxInitExtensionModule() and CDynLinkLibrary to see what these routines do with AFX_EXTENSION_MODULE and how they relate to the topic of resources.

Inside AfxInitExtensionModule()

Listing 10-5 shows the AfxInitExtensionModule() pseudocode.

Listing 10-5. The AfxInitExtensionModule() pseudocode, from DLLINIT.CPP

```

BOOL AFXAPI AfxInitExtensionModule(AFX_EXTENSION_MODULE& state, HMODULE
hModule)
{
// only initialize once
if (state.bInitialized){
    AfxInitLocalData(hModule);
    return TRUE;
}
state.bInitialized = TRUE;
// save the current HMODULE information for resource loading
state.hModule = hModule;
state.hResource = hModule;
// save the start of the runtime class list
AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
state.pFirstSharedClass = pModuleState->m_classList.GetHead();
pModuleState->m_classList.m_pHead = pModuleState->m_pClassInit;
// save the start of the class factory list
state.pFirstSharedFactory = pModuleState->m_factoryList.GetHead();
pModuleState->m_factoryList.m_pHead = pModuleState->m_pFactoryInit;
return TRUE;
}

```

First, AfxInitExtensionModule() determines if this extension DLL has been initialized before by checking the AFX_EXTENSION_MODULE state argument's bInitialized field. If the module has been initialized before, AfxInitLocalData() is called to update the module handle used by the TLS. If the DLL has not been initialized before, AfxInitExtensionModule() goes ahead and sets state.bInitialized to TRUE and then starts the initialization process.

Next, AfxInitExtensionModule() sets the hModule and hResource arguments of the state argument to the hModule argument. Then AfxInitExtensionModule() retrieves the current module state into pModuleState and gets the head of the module

state CRuntimeClass list. The result is placed in state.pFirstSharedClass. Next, the head of the module state class list is set to point to the module state class init.

Next, AfxInitExtensionModule() has similar logic for the module object factory list. The head of the object factory list is stored in the module state and then set to the m_pFactoryInit pointer in the module state.

What is MFC doing here? When the _AFX_CLASSINIT structures initialize (these are the helpers that put CRuntimeClass objects in a list), they put them in a global list in the module state. When there is an extension DLL, Microsoft wants to “transfer” that list out of the global list and into the CDynLinkLibrary object. Microsoft cannot perform the transfer without keeping track of the head of the original list. Keeping track of the head is the job of the AFX_EXTENSION_MODULE members. They need to keep track of the head of the list in case another process runs and needs to have a similar CDynLinkLibrary in its list because CDynLinkLibrary objects are all process-local.

Admittedly, what’s going on here is not obvious at first. Bear with us for a couple more sections and you will see how it all fits together. The next piece of the puzzle is the CDynLinkLibrary helper class.

Inside CDynLinkLibrary

CDynLinkLibrary is a frustratingly undocumented class (if you’ve ever written an MFC extension DLL, you know what we mean). While the MFC documentation tells you to create an instance of one, Microsoft never tells you what the class does or why.

Listing 10-6 contains the declaration for CDynLinkLibrary. Notice that CDynLinkLibrary is derived from CCmdTarget. This is kind of a mystery because CDynLinkLibrary does not appear to take advantage of its rich lineage. In other words, it does not seem to be set up to handle any messages.

Maybe the MFC team decided it is feasible to have a situation where a nonwindow/document-related message might need to be dispatched to an extension DLL such as a command message. Theoretically you have dynamically loaded command handlers. The message is left as a reader exercise to experiment and apply this nugget of MFC trivia to a real world problem.

Listing 10-6. The CDynLinkLibrary declaration, from AFXDLL_.H

```
class CDynLinkLibrary : public CCmdTarget
{
    DECLARE_DYNAMIC(CDynLinkLibrary)
public:
    // Constructor
    CDynLinkLibrary(AFX_EXTENSION_MODULE& state, BOOL bSystem = FALSE);
    // Attributes
```

```

HMODULE m_hModule;
HMODULE m_hResource;           // for shared resources
CTypedSimpleList<CRuntimeClass*> m_classList;
CTypedSimpleList<COleObjectFactory*> m_factoryList;
BOOL m_bSystem;               // TRUE only for MFC DLLs
// Implementation
public:
    CDynLinkLibrary* m_pNextDLL;      // simple singly linked list
    virtual ~CDynLinkLibrary();
};

}

```

Now we can see that the second argument to CDynLinkLibrary indicates if the extension DLL is a system DLL. Look back at the two lines of code from DllMain() in Listing 10-4 and notice that MFC sets this argument to TRUE. In the MFC documentation, they tell you to set the argument to FALSE. The end result is that when MFC is an extension DLL, it is considered a “system” DLL, and when any other extension DLL is initialized, it is considered nonsystem.

Before we look at the implications of a system versus a nonsystem DLL, notice that CDynLinkLibrary has much of the information that we found in the AFX_EXTENSION_MODULE structure. The only big addition is the m_pNextDLL CDynLinkLibrary pointer.

Let's look at the CDynLinkLibrary constructor for more answers.

CDynLinkLibrary::CDynLinkLibrary()

Listing 10-7 contains the CDynLinkLibrary constructor pseudocode.

Listing 10-7. The CDynLinkLibrary constructor pseudocode, from DLLINIT.CPP

```

CDynLinkLibrary::CDynLinkLibrary(AFX_EXTENSION_MODULE& state, BOOL
bSystem)
{
    m_factoryList.Construct(offsetof(COleObjectFactory, m_pNextFactory));
    m_classList.Construct(offsetof(CRuntimeClass, m_pNextClass));
    // copy info from AFX_EXTENSION_MODULE
    struct m_hModule = state.hModule;
    m_hResource = state.hResource;
    m_classList.m_pHead = state.pFirstSharedClass;
    m_factoryList.m_pHead = state.pFirstSharedFactory;
    m_bSystem = bSystem;
    // insert at the head of the list (extensions will go in front of
    // core DLL)
    AFX_MODULE_PROCESS_STATE* pState = AfxGetModuleProcessState();
    AfxLockGlobals(CRIT_DYNLINKLIST); pState->m_libraryList.AddHead(this);
    AfxUnlockGlobals(CRIT_DYNLINKLIST);
}

```

First, the constructor constructs the `m_factoryList` and the `m_classList`. Then the constructor copies over the information from the `AFX_EXTENSION_MODULE` state argument that describes the current MFC extension DLL into the corresponding `CDynLinkLibrary` data members.

Finally, the `CDynLinkLibrary` constructor adds itself to the current process state. The addition to the process state is locked, so that the order of DLL additions is maintained correctly. Next we'll see how the state list is traversed.

Inside `AfxFindResourceHandle()`

The end result from all of these helpers is that a list of `CDynLinkLibrary` objects (one for each extension DLL) is maintained in the module state. The extension DLL passes information about itself on to the `CDynLinkLibrary` constructor, which adds itself to the module list. Now that you know all of this background information, we can finally look at `AfxFindResourceHandle()` to see how it searches for resources. Listing 10-8 contains the pseudocode for `AfxFindResourceHandle()`.

Listing 10-8. The `AfxFindResourceHandle()` pseudocode, from `DLLINIT.CPP`

```
HINSTANCE AFXAPI AfxFindResourceHandle(LPCTSTR lpszName, LPCTSTR lpszType)
{
    HINSTANCE hInst;
    // first check the main module state
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    if (!pModuleState->m_bSystem){
        hInst = AfxGetResourceHandle();
        if (::FindResource(hInst, lpszName, lpszType) != NULL)
            return hInst;
    }
    // check for non-system DLLs in proper order
    AFX_MODULE_PROCESS_STATE* pState = AfxGetModuleProcessState();
    AfxLockGlobals(CRIT_DYNLINKLIST);
    for (CDynLinkLibrary* pDLL = pState->m_libraryList; pDLL != NULL;
        pDLL = pDLL->m_pNextDLL){
        if (!pDLL->m_bSystem && pDLL->m_hResource != NULL &&
            ::FindResource(pDLL->m_hResource, lpszName, lpszType) != NULL){
            AfxUnlockGlobals(CRIT_DYNLINKLIST);
            return pDLL->m_hResource;
        }
    }
    AfxUnlockGlobals(CRIT_DYNLINKLIST);
    // check language specific resource next
    hInst = pState->m_appLangDLL;
    if (hInst != NULL && ::FindResource(hInst, lpszName, lpszType) != NULL)
        return hInst;
    // check the main system module state
```

```

pModuleState = AfxGetModuleState();
if (pModuleState->m_bSystem){
    hInst = AfxGetResourceHandle();
    if (::FindResource(hInst, lpszName, lpszType) != NULL)
        return hInst;
}
// check for system DLLs in proper order
AfxLockGlobals(CRIT_DYNLINKLIST);
for (pDLL = pState->m_libraryList; pDLL != NULL;
     pDLL = pDLL->m_pNextDLL){
    if (pDLL->m_bSystem && pDLL->m_hResource != NULL &&
        ::FindResource(pDLL->m_hResource, lpszName, lpszType) != NULL)
        AfxUnlockGlobals(CRIT_DYNLINKLIST); return pDLL->m_hResource;
}
AfxUnlockGlobals(CRIT_DYNLINKLIST);
// if failed to find resource, return application resource
return AfxGetResourceHandle();
}

```

As you can see from Listing 10-8, AfxFindResourceHandle() uses the good old-fashioned trial-and-error method to find resources. The path it chooses is pretty important to defining the MFC extension DLL behavior.

First, AfxFindResourceHandle() looks in the current module's resource handle (returned by AfxGetResourceHandle()) for the resource if the current module is not a system module.

Next, AfxFindResourceHandle() iterates through the CDynLinkLibrary list from the process state m_libraryList. If the extension DLL is not a system DLL, FindResource() is called to try to find the resource. AfxFindResourceHandle() then looks for the information in the language-specific DLL pointed to in the process state.

Redundant Code

Can you find the redundant call in Listing 10-8?

Yep—you guessed it! AfxGetModuleState() is called twice and only needs to be called once. Microsoft has been alerted, so keep an eye on future MFC releases to see if it goes away.

Next, AfxFindResourceHandle() checks to see if the current module is a system module. If it is, AfxFindResourceHandle() calls FindResource() to search for the resource. Finally, AfxFindResourceHandle() iterates through the CDynLinkLibrary list again and checks the system extension DLLs for the resource.

In the event that none of these searches turns up anything, AfxFindResourceHandle() returns AfxGetResourceHandle(), which just returns the current resource handle for the process. AfxGetResourceHandle() is an excellent example of how MFC accesses

the various state structures. It also illustrates the difference between system and nonsystem DLLs.

Now that we've seen how MFC DLLs handle resources, let's learn more about how MFC initializes itself when running as an extension DLL.

Extension DLL Initialization and Cleanup

In the old Win16 days, a DLL had to have both a LibMain() and a WEP() (a Windows exit procedure). The LibMain() takes care of initializing the DLL, and the WEP() takes care of cleaning up. In Win32, the DllMain() routine takes care of both situations. Win32 calls DllMain() with different DWORD dwReason arguments to specify which action is taking place. The two commonly handled actions are DLL_PROCESS_ATTACH and DLL_PROCESS_DETACH. Although the MFC DllMain() makes for a pretty interesting read (it's in DLLINIT.CPP), we've pretty much covered all of the DLL-specific initialization that takes place. The rest of the code performs various error checks and bulletproofing.

Another interesting aspect of extension DLLs is some trickery that takes place in the beloved MFC message maps.

AFXDLL and Macros

True Confession time. Earlier, when we covered both message map and CObject macros, we hid some DLL-specific details from you in the macro declarations (we didn't want you getting distracted, really).

MFC has two different versions of many macros. We'll use the DECLARE_DYNAMIC() macro as an example. Listing 10-9 contains both versions of DECLARE_DYNAMIC.

Listing 10-9. Both the DLL and the non-DLL versions of DECLARE_DYNAMIC, from AFX.H

```
#ifdef _AFXDLL
#define DECLARE_DYNAMIC(class_name) \
protected: \
static CRuntimeClass* _GetBaseClass(); \
public: \
static AFX_DATA CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
#else // !_AFXDLL
#define DECLARE_DYNAMIC(class_name) \
public: \
static AFX_DATA CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
#endif //end !_AFXDLL
```

In Listing 10-9, the top DECLARE_DYNAMIC is used in DLLs and the bottom version is used in non-DLL builds. The only difference between the two is the addition of these two lines in the generated output:

```
protected:  
static CRuntimeClass * _GetBaseClass();
```

Let's look at IMPLEMENT_DYNAMIC to see if it's different, too. Listing 10-10 contains the pseudocode. (Note that IMPLEMENT_DYNAMIC calls through to _IMPLEMENT_RUNTIMECLASS, which we've also provided, for your reference.)

Listing 10-10. The IMPLEMENT_DYNAMIC declaration with DLL and non-DLL versions, from AFX.H

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \  
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)  
#ifdef _AFXDLL  
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema,  
pfnNew) \  
    CRuntimeClass* class_name::_GetBaseClass() \  
    { return RUNTIME_CLASS(base_class_name); } \  
    AFX_DATADEF CRuntimeClass class_name::class##class_name = { \  
        #class_name, sizeof(class class_name), wSchema, pfnNew, \  
        &class_name::_GetBaseClass, NULL }; \  
    static const AFX_CLASSINIT \  
        _init_##class_name(&class_name::class##class_name); \  
    CRuntimeClass* class_name::GetRuntimeClass() const \  
    { return &class_name::class##class_name; } \  
#else  
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema,  
pfnNew) \  
    AFX_DATADEF CRuntimeClass class_name::class##class_name = { \  
        #class_name, sizeof(class class_name), wSchema, pfnNew, \  
        RUNTIME_CLASS(base_class_name), NULL }; \  
    static const AFX_CLASSINIT \  
        _init_##class_name(&class_name::class##class_name); \  
    CRuntimeClass* class_name::GetRuntimeClass() const \  
    { return &class_name::class##class_name; } \  
#endif
```

In Listing 10-10, the plot thickens! The DLL and non-DLL versions are significantly different now. First, the DLL version generates this member function:

```
CRuntimeClass * CMyClass::_GetBaseClass()  
{
```

```

    return RUNTIME_CLASS(CMyBaseClass);
}
}

```

Also, notice that when `IMPLEMENT_DYNAMIC` spits out the initialization code that will eventually end up initializing the static `CRuntimeClass` object (see Chapter 5 for all the details of these macros; we're focusing on the DLL-specific issues here), it passes the address of the `_GetBaseClass` function and not just `RUNTIME_CLASS(CMyBaseClass)`, like the non-DLL version.

As you would imagine, the `CRuntimeClass` structure allows for this difference with the following `ifdef`:

```

#ifndef _AFXDLL
    CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
#else
    CRuntimeClass* m_pBaseClass;
#endif

```

Remember that `RUNTIME_CLASS` returns the address of the static `CRuntimeClass` that is embedded in your class with the `DECLARE_DYNAMIC` macro.

To summarize, the difference between DLL and non-DLL cases is that MFC has to call a function that returns the address of the static `CRuntimeClass` member instead of actually storing and accessing the address of the static `CRuntimeClass` data member.

Why does MFC have to go through this level of indirection for DLL builds? We asked our MFC internals Microsoft contact, Dean McCrory who responds:

Exported/imported data in Win32 has to be initialized by the C-runtime before being used. This has to do with the implementation of exported data on win32; there is an extra level of indirection and basically the pointer has to be initialized at startup time. The pointer is initialized by the C-runtime at the time of object construction. It actually pretends that a class like `CRuntimeClass` has a constructor, even though it doesn't and appears to be initialized statically at compile time. Because this feature is handled by constructing the object with normal static object construction mechanisms, the order in which the `CRuntimeClass` objects are constructed is not predictable with respect to other static objects. Thus, we discovered that if we just imported the data directly, the user could write code (for example, a constructor for an object instantiated globally) that would access uninitialized `CRuntimeClass` structures! To work around this problem, we get at the exported data through a function, which works without any special initialization. This makes the code a bit smaller (no special construction of the `CRuntimeClass` objects) and fixes the problem of accessing the `CRuntimeClass` objects early during static object initialization.

You will notice that almost all of the MFC macros have AFXDLL and non-AFXDLL versions, including the message map macros.

Now that you know more about MFC extension DLLs than most of the people actually using this MFC feature, it's time to move on and look at MFC threads.

MFC Threads

In this section, we look at how MFC implements its two thread encapsulations: worker threads and user-interface threads. As usual, we will be sure to point out any interesting Win32 APIs in action. However, this section assumes that you are already thread literate, so you may want to brush up by visiting your local advanced Win32 programming book or spending some quality time with the Visual C++ books on-line.

Also, we'll see the MFC state classes peppered throughout the thread code that we examine. If you've jumped here from the index or table of contents, you may want to check out the first section of this chapter before going on. It has a primer on the MFC states.

As we already said, MFC supports two different types of threads: worker threads and user-interface threads. Because threads is already a pretty complex topic, we'll start out with the easier of the two, worker threads, and then ramp up to user-interface threads.

The MFC user starts both types of threads by calling `AfxBeginThread()`. There are actually two overrides of `AfxBeginThread()`. In the worker-thread case, you pass a `CWinThread` object and a pointer to a controlling function (the controlling function does the work). For a user-interface thread, you need to create a `CWinThread` override and pass its `CRuntimeClass` information to `AfxBeginThread()`.

Both versions of `AfxBeginThread()` take some common arguments that you would expect are needed for thread creation, such as thread priority and security attributes.

Now let's dive right into the thick of it and see how MFC implements worker threads.

MFC Worker Threads

Let's start by looking at how the worker-thread version of `AfxBeginThread()` is implemented. Listing 10-11 contains the pseudocode for `AfxBeginThread()`.

Listing 10-11. `AfxBeginThread()`—worker-thread variant, from `THRDCORE.CPP`

```
CWInThread* AFXAPI AfxBeginThread(AFX_THREADPROC pfnThreadProc,
    LPVOID pParam, int nPriority, UINT nStackSize, DWORD dwCreateFlags,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs)
```

```

{
    CWinThread* pThread = new CWinThread(pfnThreadProc, pParam);
    if (!pThread->CreateThread(dwCreateFlags|CREATE_SUSPENDED,
        nStackSize, lpSecurityAttrs)) {
        pThread->Delete();
        return NULL;
    }
    VERIFY(pThread->SetThreadPriority(nPriority));
    if (!(dwCreateFlags & CREATE_SUSPENDED))
        VERIFY(pThread->ResumeThread() != (DWORD)-1);
    return pThread;
}

```

First, AfxBeginThread() instantiates a CWinThread object on the heap and passes in the worker function pointer as argument, along with the pParam (the argument that's passed to your worker function).

Next, AfxBeginThread() uses the CWinThread object to spin an initially suspended thread (CREATE_SUSPENDED) and passes in the thread property arguments to CWinThread::CreateThread(). If the CWinThread::CreateThread() call fails, the CWinThread object is deleted and NULL is returned.

After calling CWinThread::CreateThread(), AfxBeginThread() sets the priority by calling CWinThread::SetThreadPriority(). Finally, unless the caller set the CREATE_SUSPENDED flag, AfxBeginThread() sets the thread in motion by calling CWinThread::ResumeThread(). A pointer to the new CWinThread object is returned from AfxBeginThread().

All Roads Lead to CWinThread

Well, one thing is obvious from looking at AfxBeginThread(): the MFC CWinThread class is key to understanding the MFC thread implementation. So let's crack it wide open! Listing 10-12 contains the more interesting parts of the CWinThread declaration. We have chopped out most of the documented member functions, so if you are not familiar with them, you may want to take a look in the on-line documentation.

Listing 10-12. The abbreviated CWinThread declaration, from AFXWIN.H

```

class CWinThread : public CCmdTarget
{
DECLARE_DYNAMIC(CWinThread)
public:
// Constructors *omitted
// Attributes
    CWnd* m_pMainWnd; // main window (usually same AfxGetApp()->m_pMainWnd)
    CWnd* m_pActiveWnd; // active main window (may not be m_pMainWnd)
    BOOL m_bAutoDelete; // enables 'delete this' after thread termination

```

```

HANDLE m_hThread;           // this thread's HANDLE
DWORD m_nThreadID;         // this thread's ID

// Operations *omitted
// Overridables *omitted
// Implementation

public:
    void CommonConstruct();
    virtual void Delete();
    MSG m_msgCur;           // current message

public:
    // constructor used by implementation of
    AfxBeginThread CWinThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam);
    // valid after construction
    LPVOID m_pThreadParams; // generic parameters passed to starting
                            // function
    AFX_THREADPROC m_pfnThreadProc;

protected:
    CPoint m_ptCursorLast;   // last mouse position
    UINT m_nMsgLast;         // last mouse message
};

}

```

After we recap what each of the implementation-specific members do, we will look at a couple of key CWinThread function implementations to see what's going on inside this powerful MFC class. First, notice that CWinThread is a CCmdTarget, which indicates that it gets involved in message routing. (More on this later.)

CWinThread Data Members

- **m_pMainWnd**—A pointer to the application's main window. CWinThread needs this pointer to be able to route messages correctly.
- **m_pActiveWnd**—A pointer to the main window of the container application when an OLE server is in-place active.
- **m_bAutoDelete**—When TRUE, CWinThread will kill itself when the thread terminates.
- **m_hThread**—The handle of the Win32 thread encapsulated by CWinThread. After verifying that this is active, you can use it in Win32 thread APIs if you want. Later we'll see how it is set up and used internally by CWinThread.
- **m_hThreadID**—The Win32 thread ID for the thread encapsulated by CWinThread.
- **m_msgCur**—Buffers the current message being processed by the CWinThread message pump. (More on this later.)
- **m_pThreadParams**—Stores the pParam argument, which gets passed to your worker function.

- `m_pfnThreadProc`—A pointer to the worker function.
- `m_ptCursorLast`—The CPoint for the last mouse move message. This is used to filter out redundant mouse move messages in the CWinThread idle processing.
- `m_nMsgLast`—Similar to `m_ptCursorLast`; used to detect two sequential messages. (For example, current and last are equal.)

CWinThread Implementation Member Functions

- `CommonConstruct()`—Both constructors (worker and ui thread) eventually call into here. We won't look at the pseudocode, because `CommonConstruct()` just sets the data members to sane defaults.
- `Delete()`—If `m_bAutoDelete` is true, `Delete()` calls “delete this” and thus causes the thread to kill itself. (Suicide!)

Now let's see how threads actually get created inside MFC by looking into `CWinThread::CreateThread()`.

Thread Creation—Inside CWinThread::CreateThread()

`CreateThread()` is complex enough that we'll need to take it in two pieces. First, the thread is created using some internal MFC utility structures and functions. Second, MFC performs some interesting initializations on both the thread and CWinThread.

When MFC creates a Win32 thread, it calls the `_beginthreadex()` run-time library routine, which takes several arguments. The most important for our discussion are these:

- Third argument—A pointer to the routine (start address) of the routine that begins the execution of the new thread. In the `CWinApp::CreateThread()` case, this is `_AfxThreadEntry`.
- Fourth argument—A pointer to an argument which is passed to the thread routine (see above). MFC defines an internal structure, `_AFX_THREAD_STARTUP`. We'll see these used soon.
- Sixth argument—The address of a new thread. `CWinThread::CreateThread()` passes in the address of `m_nThreadId`, which gets set to the thread address (ID).

Let's look at the first piece of `CWinThread::CreateThread()` with this sneak preview information in mind. Listing 10-13 contains the pseudocode for `CWinThread::CreateThread()`.

Listing 10-13. The first piece of an abbreviated CWinThread::CreateThread(), from THRDCORE.CPP

```
BOOL CWinThread::CreateThread(DWORD dwCreateFlags, UINT nStackSize,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs)
```

```

{
    // setup startup structure for thread initialization
    _AFX_THREAD_STARTUP startup;
    memset(&startup, 0, sizeof(startup));
    startup.pThreadState = AfxGetThreadState();
    startup.pThread = this;
    startup.hEvent = ::CreateEvent(NULL, TRUE, FALSE, NULL);
    startup.hEvent2 = ::CreateEvent(NULL, TRUE, FALSE, NULL);
    startup.dwCreateFlags = dwCreateFlags;
    // **some event checking and cleanup omitted for brevity.
    // create the thread (it may or may not start to run)
    m_hThread = (HANDLE)_beginthreadex(lpSecurityAttrs, nStackSize,
        &_AfxThreadEntry, &startup, dwCreateFlags | CREATE_SUSPENDED,
        (UINT*)&m_nThreadID);
    if (m_hThread == NULL)
        return FALSE;
}

```

CreateThread() starts off by creating a local `_AFX_THREAD_STARTUP` and storing valuable information about the thread, which is passed to the thread routine during thread execution. This information includes the following:

- `pThreadState`—Our good old `_AFX_THREAD_STATE` from Listing 10-3.
- `pThread`—A back pointer to the `CWinThread`.
- `hEvent/hEvent2`—Two event handles for internal `CWinThread` “triggering.” `hEvent` is triggered after thread creation succeeds. `hEvent2` is triggered after a thread is resumed to jump-start it.
- `dwCreateFlags`—The creation flags that specify if the thread should be suspended, and so on. The thread may need to check these later, so they are stored.

Initialization-only information, such as the security attributes, is not stored in the `_AFX_THREAD_STARTUP`, because the thread either doesn’t need it while running or can access it by going through the `pThread` pointer.

After initializing the local `_AFX_THREAD_STARTUP` structure with this information, CreateThread() creates a thread by calling `_beginthreadex()`—just as we predicted! CreateThread() passes in the security, stack size, `_AfxThreadEntry`, `&startup`, creation flags, and the address of the thread ID.

Unless one of the arguments is bogus, or the system cannot spin a new thread because of some resource problem (such as no memory), a thread is born!

Remember, CreateThread() called `_beginthreadex()` with `CREATE_SUSPENDED`, so though the new thread is created, it is in the deep freeze until MFC decides to thaw it out.

The remaining pseudocode for CreateThread() is in Listing 10-14.

Listing 10-14. An abbreviated CWinThread::CreateThread(), after the thread is created, from THRDCORE.CPP

```
//CWinThread::CreateThread() continued...
ResumeThread();
::WaitForSingleObject(startup.hEvent, INFINITE);
::CloseHandle(startup.hEvent);
// if created suspended, suspend it until resume thread wakes it up
if (dwCreateFlags & CREATE_SUSPENDED)
    ::SuspendThread(m_hThread);
if (startup.bError)
//**Error cleanup omitted - lots of frees and closes <g>
// allow thread to continue, once resumed (it may already be resumed)
::SetEvent(startup.hEvent2); return TRUE;
}
```

Here's where things get a little wild and woolly, so hold onto your seat and put on your thinking cap. CreateThread() calls ResumeThread() (which calls ::ResumeThread(m_hThread)).

At this point the thread we created starts chugging. This is a very fragile time in the life of our thread. It doesn't yet have an _AFX_THREAD_STATE pointer, an m_pMainWnd pointer, and several other important items that every true MFC thread should have.

Also, we now really have two threads! There's a thread that originally called CreateThread() (the mom thread) and then there's our toddler thread.

Now CreateThread() puts the brakes on the mom thread by making it wait on the _AFX_THREAD_START::hEvent. And now toddler thread takes off! When it does, it starts right where the CreateThread() _beginthreadex() call told it to start: _AfxThreadEntry().

_AfxThreadEntry() is a seriously long routine, but we can't abbreviate it much without losing the context of what it's doing. Here's a blow-by-blow of what happens to our toddler thread when it starts running through _AfxThreadEntry():

1. Right off the bat, _AfxThreadEntry() (abbreviated _ATE from now on), takes its _AFX_THREAD_STARTUP argument and stores the pThread field in a bona fide CWinThread pointer, so it doesn't have to muck around with the _AFX_THREAD_STARTUP pointer all the time.
2. Now _ATE calls AfxGetThreadState(), which returns mom thread's state. The toddler thread uses the mom thread's state intact for the most part. The only field that it changes is the module state, which it gets from the _AFX_THREAD_STARTUP argument. The new toddler thread "inherits" the module state from the mom thread. (You may want to look at the code in THRDCORE.CPP to see how this works firsthand.)

3. Next, `_ATE` calls `AfxInitThread()`, which does still more state manipulations. The difference is that `AfxInitThread()` sets up a message queue for the thread (`SetMessageQueue`) and sets up some message filters. (Remember the message filter handles back in Listing 10-3?)
4. After `AfxInitThread()`, `_ATE` creates a local `CWnd` object and attaches it to the current main window by attaching it to the `CWinApp::m_pMainWnd::m_hWnd` and then setting the `CWinThread::m_pMainWnd` pointer to point to its address.
5. At this point, toddler thread has aged before our eyes and is now adolescent thread! In its last moments before adulthood, when mom thread pushes it out of the nest and into the harsh realities of thread life, the `hEvent2` handle is copied out of the `_AFX_STARTUP_THREAD` pointer because that pointer is about to become toast.
6. At long last, adolescent thread is ready to become an adult thread and signals mom (remember, she's been waiting patiently all this time—thanks, Mom!) that it is an adult. The signal is to set the event, which causes the `::WaitForSingleObject()` in the mom to resume. The actual call is `::SetEvent(pStartup->hEvent)`.
7. Now the new adult thread is ready for action, but there is one more operation that mom may want to perform before finally saying goodbye. `_ATE` calls `::WaitForSingleObject(hEvent2, INFINITE)` to put the new adult thread on ice until mom gives it the final green flag.

At this point we're back to mom thread and thus back to Listing 10-14, after the `::WaitForSingleObject()` call. Mom thread cleans up the `hEvent` handle for junior by calling `::CloseHandle()` and then calls `::SuspendThread()` on junior if the caller specified that the thread should be created suspended.

Now that mom has made sure that the specified suspended state is satisfied, she sends junior thread on his way by calling `SetEvent` on the second event.

Wait! This is where it starts to get really interesting. At this point, mom goes on her merry way, but junior is either suspended in or chugging through `_ATE`. Let's assume that he's running through there and pick up `_ATE` where we left off.
8. After the `WaitForSingleObject()`, junior thread says his final goodbye to mom thread and closes the handle on `hEvent2`.
9. Now, if the `CWinThread::m_pfnThreadProc` is non-NULL, junior thread realizes that he was born to be a worker thread and starts working by passing control over to `m_pfnThreadProc`. Remember that we passed in this pointer way back in the call to `AfxBeginThread()` and until now it has not seen the light of day!
10. If junior thread's `m_pfnThreadProc` pointer is NULL, he realizes that his calling in life is to be a user-interface thread.

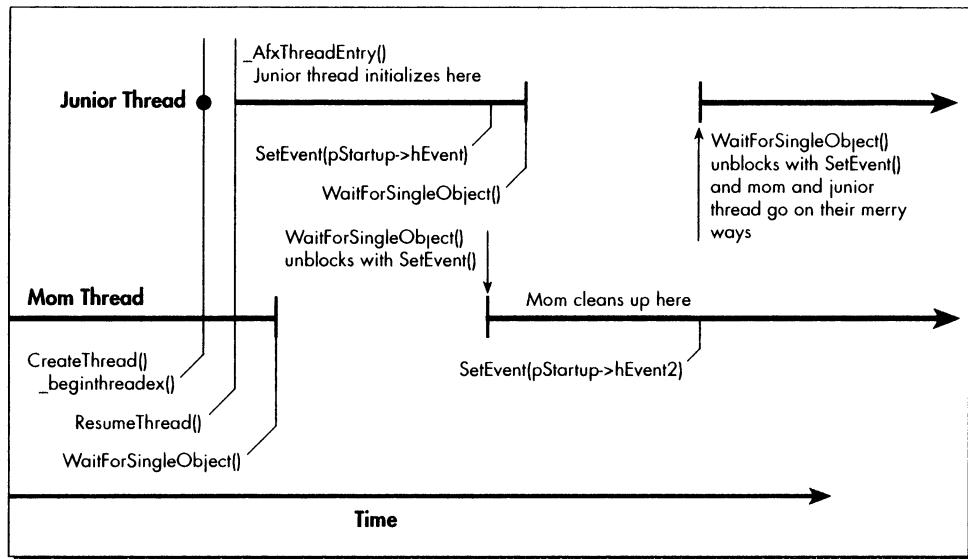


Figure 10-2. A time line for mom and junior thread

We leave junior thread hanging in `_AfxThreadEntry()` to introduce you to some user-interface thread concepts before we go on.

Figure 10-2 illustrates what has happened to our two threads, so that you can get a clearer mental image.

MFC User-Interface Threads

Worker threads are very single-minded: they have one function to run and that's about it. User-interface (UI) threads have to be much more savvy and do things such as responding to user events and handling user input.

Where a worker thread had to just provide `CWinThread` with a pointer to a function, a user-interface thread has to derive from `CWinThread` and provide several member function overrides to work properly. The overrides are these:

- `ExitInstance()`—Performs cleanup when the thread terminates.
- `InitInstance()`—Performs thread-instance initialization.
- `OnIdle()`—Performs thread-specific idle-time processing.
- `PreTranslateMessage()`—Filters messages.
- `Run()`—A message pump. (This is where you're non-MFC's message loop lives now.)

CWinApp: The Ultimate UI Thread!

Sound familiar? Yep, you guessed it: CWinApp is a user-interface thread. It's up to your CWinApp derivative to provide a useful `InitInstance()`. CWinThread provides the default `Run()` and `OnIdle()` routines. The rest of CWinApp's job is to interact with the user, which it does very well.

We will visit both `Run()` and `OnIdle()` in detail, but before we do, let's continue with the story of junior thread. When we last saw junior thread, it had realized that it was a user-interface thread and not a worker thread by checking its `m_pfnThreadProc`.

Back to `_AfxThreadEntry()` and Junior Thread

Once junior thread realizes that it is a user-interface thread, it starts the user-interface thread process. Listing 10-15 shows the user-interface-specific portion of `_AfxThreadEntry()`.

Listing 10-15. The user-interface thread-specific portion of `_AfxThreadEntry()`, from `THRDCORE.CPP`

```
//Initialization and synchronization with mom before here.
if (pThread->m_pfnThreadProc != NULL)
    nResult = (*pThread->m_pfnThreadProc)(pThread->m_pThreadParams);
else if (!pThread->InitInstance())
    nResult = pThread->ExitInstance();
else
    nResult = pThread->Run();
// cleanup and shutdown the thread
threadWnd.Detach();
AfxEndThread(nResult);
return 0; // not reached
```

In Listing 10-15, first we see the check for `m_fnThreadProc` and `_ATE` calling `m_pfnThreadProc` to start the worker thread working. In the “else if” statement, there is a call to `CWinThread::InitInstance()`, which should be overridden by the user-interface `CWinThread` derivative to do something and not return FALSE. If it does call FALSE, `ExitInstance` is called and the thread exits.

Next, if `InitInstance()` returns TRUE, `CWinThread::Run()` is called. This usually is not overridden by the user-interface `CWinThread` derivative, so the actual `CWinThread::Run()` is called. (You are free to override `Run()`, but there just doesn't seem to be very many cases when you would want to.)

We'll investigate `Run()` in a minute, but as the name implies, this kicks the thread into a mode where it basically hangs out and responds to events.

Next, `_ATE` has cleanup code that is executed when either the worker thread's `m_pfnThreadProc` returns, `InitInstance()` returns FALSE, or `Run()` returns. The cleanup code detaches the local `CWnd` main window object and calls `AfxEndThread()`, which does more cleanup and eventually calls `_endthreadex()` (the antithesis of `_beginthreadex()`).

Now you know the full story of the life and times of both worker and user-interface threads. However, there are some holes in the story. The biggest hole is what happens during junior's run stage—the prime of his life.

Run, Thread. Run!

Way back in Chapter 2 we promised that we would cover CWinThread::Run(), and now your persistence has paid off. We've actually come full circle in our coverage of MFC's GUI classes.

If we had to vote, CWinThread::Run() would be the single most important block of code in MFC. Nothing else in MFC defines how your MFC application works more than these 30 lines of code. We've seen countless questions about idle processing, message processing, and others. All of them are answered by CWinThread::Run(), which appears in Listing 10-16.

Listing 10-16. CWinThread::Run()—The most important 30 lines in MFC, from THRDCore.cpp

```
int CWinThread::Run()
{
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;
    for (;;) {
        // phase1: check to see if we can do idle work
        while (bIdle &&
               !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE)) {
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }
        // phase2: pump messages while available
        do {
            if (!PumpMessage())
                return ExitInstance();
            if (IsIdleMessage(&m_msgCur)) {
                bIdle = TRUE;
                lIdleCount = 0;
            }
        } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
    }
}
```

First, be warned: Run() is only 30 lines of code, but it is deceptively complex. There is more going on here than meets the eye at first glance. Read on . . .

CWinThread::Run() is an infinite loop that breaks into two distinct phases:

- Phase 1—A “while” loop that “idles” while there are no messages in the message queue for the thread (in other words, ::PeekMessage() returns FALSE).
- Phase 2—A “do-while” loop that “pumps” while there are messages to be pumped (in other words, ::PeekMessage() returns TRUE).

The two loops are “connected” by the bIdle flag and the lIdleCount LONG variables.

Let’s examine the two phases in more detail.

Idle Processing

At the start of the day, bIdle is TRUE and lIdleCount is 0. bIdle will always be TRUE until the ::PeekMessage() returns FALSE, indicating that there are no pending messages on the queue and the thread enters the “while” block of phase 1. In this block, a call to OnIdle() is made.

OnIdle() is called by Run() to indicate to the thread that the user is not performing any actions, so the thread can do some cleanup work or any other work that it may want to perform. When the thread is done with its idle processing, it returns 0. The argument to OnIdle() can be used to determine how long the thread has been inactive, so that OnIdle() can prioritize any idle processing. High-priority idle processing would happen after low counts and low-priority (lengthy) idle processing after high counts of inactivity.

OnIdle() can be the default CWinThread::OnIdle(), which is directly called by CWinApp::OnIdle(). Or if you override CWinApp::OnIdle() in your CWinApp derivative, it will be your OnIdle(), plus you have to call CWinApp::OnIdle(). Regardless of whether you choose to have idle processing, CWinThread::OnIdle() is called.

Listing 10-17 contains some very pseudo pseudocode for CWinThread::OnIdle().

Listing 10-17. A very abbreviated CWinThread::OnIdle(), from THRDCore.cpp

```
BOOL CWinThread::OnIdle(LONG lCount)
{
    if (lCount <= 0) {
        // send WM_IDLEUPDATECMDUI to the main window
        // **omitted
        // send WM_IDLEUPDATECMDUI to all frame windows
        // **omitted
    }
    else if (lCount >= 0) {
        // Call AfxLockTempMaps/AFxUnlockTempMaps
        // to free maps, DLLs, etc..
        // ** code omitted
    }
    return lCount < 0; // nothing more to do if lCount >= 0
}
```

Though we had to omit major blocks of code from Listing 10-17, you can still see the flow of control through `CWinThread::OnIdle()`. Notice that if you call `OnIdle(-1)` it forces a UI update. This can be handy if you are waiting on some lengthy process (serial I/O—Database access) and want to update the UI components of your MFC application while waiting.

`CWinApp::OnIdle()` (in APPCORE.CPP) builds on this idle processing and adds handlers for when `lCount` is 0 or 1. Your `OnIdle()` handlers should reserve these “high-priority” counts for MFC and start idle processing at 2 or higher.

Now that we’ve learned about idle processing, let’s go back to `::Run()`. Phase 1 continues to call `OnIdle`, incrementing `lIdleCount` while there are no messages in the queue. If all idle processing is done, `bIdle` is set to FALSE and we fall down to phase 2. Also, if a message ever comes in during this process, we will fall down to phase 2 once `OnIdle()` returns after the message arrives (which gives the `PeekMessage()` in the “while” a chance to execute and return TRUE).

Message Pumping: `CWinThread::Run()` Phase 2

In phase 2, a thread enters the “do-while” loop and stays in there while there are messages on the thread’s message queue. During the “do-while” loop, the thread calls `PumpMessage()`, which causes the message to be dispatched, and then `IsIdleMessage()`. `IsIdleMessage()` determines if the current message being processed (remember this from Listing 10-12?), which is set in `PumpMessage()`, is an idle message. An idle message is basically a message sent by Windows that indicates that the user is not doing something with the current thread.

For example, a bunch of mouse move messages at different points or `WM_COMMAND` messages—a message that indicates some program state has changed—and the `OnIdle()` need to be called again (think of idle messages that kick the idle process into running). Messages like mouse move messages at the same position or paint messages do not generally change the program state (`WM_PAINT` responds to changes in the program state but usually doesn’t change it), so there is no need to call `OnIdle()` for those messages. You can override `OnIdle()` if there are other window messages that travel through the message queue that you don’t want to cause idle processing to run.

If this is the case, `bIdle` is set to TRUE (it could have been set to FALSE only if `OnIdle()` indicated that it was done processing) and zeroes out the `lIdleCount`.

Does Idle Processing Mean Busy Waiting?

One common concern about `CWinThread`’s idle processing is that the application will “busy wait”. In other words, will your MFC application be hogging the processor and Windows’ attention doing all of this idle processing.

Looking at Run(), eventually there will be either no more messages on the queue and OnIdle() will return FALSE. This causes a call to PumpMessage(), which blocks in GetMessage(). Don't be concerned about idle processing burning up those spare CPU cycles, just make sure that your OnIdle() performs very small prioritized processing and that it will return FALSE eventually.

The only way out of the infinite loop is for PostMessage() to return FALSE, which happens only when a PostQuitMessage() is called and the thread receives the WM_QUIT message. In this case, the second phase calls ExitInstance() and returns the result. Then the thread flow of control bumps back up to _AfxThreadEntry and the user-interface thread is cleaned up.

There is one final aspect of threads that we want to cover so that you'll be aware of any potential problems when writing multithreaded MFC applications.

A Tale of Threads, Handles, and Objects

One key point to remember about the MFC thread implementation is that each thread has its own message queue. This seems innocent enough, but the implications are pretty large. Think back to Chapter 2. Remember that MFC "attaches" its objects to Windows by subclassing certain messages.

Because each thread has its own message queue, each thread has its own mapping of MFC objects to Windows handles. When writing user-interface threads, keep this in mind and try to localize the threads to a small number of user-interface objects that only that thread has to access. You can get around this limitation by using the old-fashioned ::PostMessage() back and forth between threads, but that could get tricky too.

Now you know most of the key points about the MFC thread implementation. This knowledge will help you write better multithreaded applications using MFC and hopefully save you time and effort doing so. Before moving on to the next chapter, we want to recap the key points that we covered in this chapter and give you some pointers to other related areas of MFC.

Conclusion

In this chapter, we learned the following about MFC DLLs:

- Both DLL and thread implementations depend on the internal MFC state classes. The MFC states separate data into different logical boundaries and keep threads and DLLs from clobbering each other's data.

- In MFC 4.0, there are really 3 MFC DLL models:
 - Regular DLL, statically linked (formerly known as a _USRDLL)
 - Regular DLL, dynamically linked (formerly didn't exist)
 - Extension DLL, always dynamically linked (didn't really change name)

Both kinds of Regular DLLs can be called from non-MFC apps. Extension DLLs are only designed to extend an existing MFC app. That MFC app must use the MFC DLL as well. It is possible for a DLL to operate in modes #2 and #3 (MFC40.DLL is one example), but this is rare and somewhat tricky.

- In MFC, extension DLLs are built with the _AFXDLL flag.
- Extension DLLs have lists of resources and other items that are searched at run time. CDynLinkLibrary is the helper class that is key to this implementation.
- MFC has to play tricks with some of the macros that access the CRuntimeClass information so that static objects will be created correctly in extension DLLs.

And we learned these things about MFC threads:

- They can be one of two types: worker or user interface. Both are created using _beginthreadex() and end with _endthreadex().
- The MFC class CWinThread is the class that provides all of the thread support. The most important part of CWinThread is the CreateThread() member function.
- CWinThread::CreateThread() creates a thread and uses _AfxThreadEntry() to provide the path of execution for the thread.
- User-interface threads, like CWinApp, spend most of their lives in ::Run(), where they respond to messages and handle idle processing.
- The most critical lines of code in MFC that every MFC developer needs to be aware of are CWinThread::Run(). CWinThread::Run() is the heart of MFC—it pumps messages through the user-interface arteries, delivering information to and from the user.

For More Information

If you want to learn even more about the topics covered in this chapter, here are some suggested starting points:

- MFC is several DLLs, each of them with different initializations. Check out the source files DLLDB.CPP, DLLINIT.CPP, DLLMODULE.CPP, DLLNET.CPP, DLLOLE.CPP, and OLEDLL.CPP to learn more about the requirements and implementations of each DLL.

- Examine AFXTLS_.H and AFXTLS.CPP and answer these questions:
 - What does CThreadSlotData do and how?
 - Figure out what the following call does and how:
AfxLockGlobals(CRIT_DYNLINKLIST);
- Examine WINHAND_.H and WINHAND.CPP and answer these questions:
 - Why does CHandleMap have to have CWinThread as a friend?
 - Why does CHandleMap have two maps?
- Set a breakpoint in CWinApp::Run() with your debugger and follow several messages through this body of code to get a feel for how it works.
- Examine the implementation of the MFC synchronization classes in AFXMT.H, MTCORE.CPP, and MTEX.CPP and answer these questions:
 - What functionality does the CSyncObject base class provide?
 - What is the internal difference between CSingleLock and CMultiLock?
 - Where are the actual Win32 API calls for each of the synchronization classes?

Up Next

In the next chapter we start our coverage of the implementation of OLE in MFC. We start at the bottom—how MFC supports COM—and work our way up the various OLE components.

How MFC Implements COM

Visual C++'s support for OLE should be a mystery to no one. Besides the basic AppWizard support for OLE compound documents and Automation servers and the basic ControlWizard support for OLE controls, MFC provides a set of classes for adding many different OLE features to your MFC application. However, the OLE paintbrush supplied by MFC is very wide, and the support for OLE functionality is very coarse.

It's easy to include OLE features (such as container support, server support, and Automation support) at a very high level. Just crank up the AppWizard, select your options, and let it generate the code for you. Adding the code for Ole-enabling your application is straightforward after that point. *Violà!* You've got an instant compound document or Automation application. (Well, not that instant, but one heckuva lot easier than doing everything by hand.)

However, there are some times when you need to program at a lower level. For example, if you want to develop a Component Object Model (COM) class that implements a custom interface, you'll have to sidestep the standard framework support and write COM code directly. Or, if you want to derive your own class from one of MFC's OLE classes and implement another interface, you'll have to rely on some lower-level support provided by MFC.

This chapter shows how MFC implements COM. First, we'll review OLE's COM. Then we'll examine MFC's support for COM, which includes CCmdTarget, interface maps, and the COleObjectFactory, as well as some undocumented functions for implementing in-proc servers. Finally, we'll see how to write a COM class that implements a Microsoft-defined interface (IPersist) and a custom interface called IMath.

MFC and OLE

When it comes to OLE programming, MFC takes care of a great deal of the necessary “boilerplate” OLE code for you. AppWizard starts the show by inserting the raw OLE infrastructure—all you need to do is add application-specific behavior. In many ways, you can think of MFC’s relationship to OLE the same way you think of a compiler’s relationship to assembly language. You can do most of what you need by using high-level abstractions. However, you can always dip down into the assembly-language level of programming.

When writing an OLE-enabled application, you can rely on the high-level support provided by MFC to do most of the work for you. However, as with high-level languages, there are occasions when you need to get down and work at the system’s native level. In the case of OLE, that means working down with COM. Remember, MFC is set up to be an application framework—a finely tuned machine that already works—you just insert the functionality you want. This is usually adequate for implementing standard OLE features such as OLE document containers and servers, and automation servers. However, if you need to do something such as write a custom interface or implement an interface left out by MFC, then you need to work with COM at the native level.

Fortunately, MFC does provide support for working with COM. That support comes through CCmdTarget, COleObjectFactory, and MFC’s interface map macros. Inside its many OLE classes, MFC is busy implementing the interfaces necessary to make Uniform Data Transfer, OLE document applications, Automation servers, and OLE controls work. We’ll examine those classes very soon. But before we look at MFC’s support for OLE features, let’s figure out how MFC implements the Component Object Model. We’ll start by reviewing COM briefly.

The Component Object Model

OLE is the name of Microsoft’s object technology. Above all else, OLE is meant to be a consistent way of integrating software that allows software to evolve over time: it’s an integration technology. The major goal of OLE is to open up applications to the outside world. That is, the reason for programming in COM is to bring software reusability out from the source code realm and into the binary realm. For example, think about how you write applications today. You usually work in a single language (C++, of course). When you get done writing your application, it usually consists of a single executable file and perhaps an assortment of DLLs.

This works fine in many cases. However, working this way usually means that your application is sealed up: it's an entity unto itself. The only way to share functionality and data is through a variety of inconsistent, ad hoc methods like using shared DLLs or by using DDE.

COM solves this problem by introducing a consistent protocol for allowing software components to communicate. Microsoft's Component Object Model is going to play a big part in all future implementations of Windows, so it's worth understanding.

COM smooths out what Jeff Duntemann, editor of *Visual Developer* magazine, calls "the edges." That is, COM aligns applications so that they know how to talk to one another. By programming to the Component Object Model, you ensure that all the components within a system can speak to one another in a consistent manner. Current mechanisms for sharing data and functionality are inconsistent. And often times, sharing functionality requires all participating applications to have knowledge about one another and/or requires some sort of compiler-dependent layout knowledge. Because COM defines a binary standard, it's compiler and language independent. Thus, COM effectively defines a protocol for objects to communicate with each other at the binary level.

COM programming generates several issues that don't arise in regular programming. You need to become familiar with these notions to understand COM. These ideas include the following:

- Defining COM classes.
- Exposing functionality through a binary firewall.
- Identifying COM classes (using a way that persists after compile time and is language independent).
- Managing the lifetime of an object.
- Implementing run-time discoverability.
- Housing COM classes (in DLL and EXE servers).
- Using OLE classes from the client point of view.

These concepts are covered in the following sections.

What Is a COM Class?

As you learn about COM, keep one idea at the top of your thinking: COM is all about sharing software *at the binary level*. This is completely different from (yet complementary to) the ideas involved in object-oriented languages like C++.

Go back to your C++ basics for a moment. C++ classes are data structures with member functions attached to them. For example, consider a simple string class, as shown in Listings 11-1 and 11-2.

Listing 11-1. A header file for a C++ string class

```
class string {
    char m_achString[256];

public:
    string();
    char * GetString(char *);
    void SetString(char *);
    int GetLength();
};
```

Listing 11-2. A source file for a C++ string class

```
string::string() {
    m_achString[0] = '\0';
}

string::GetString(char *psz) {
    if(psz)
        strncpy(psz, m_achString, 255);
    return
        psz;
}

void string::SetString(char * psz)  {
    if(psz)
        memset(m_achString, 0, 256);
        strncpy(m_achString, psz, 255);
}
}

int string::GetLength() {
    return strlen(m_achString);
}
```

This function has a single interface (that is, all the public member functions). Given this class definition, the only thing you need to do is create an object of this class using operator new and go to town. Call the functions to your heart's content. Then when you're done, just use operator delete to get rid of it.

C++ classes are a great way to organize your code. However, they aren't conducive to sharing code at the binary level. How would you share the string class in Listings 11-1 and 11-2? You could actually export the class, but then you'd have to

deal with name-mangling issues (which are compiler dependent). If the compiler vendor decides to change the name-mangling scheme or your clients decide to use a different compiler than yours, you're hosed. You could define a set of straight C API functions and export them, but then you'd lose the advantages of working with classes. If you're using MFC, you could put the class in an AFX extension DLL. But then if you later decide to change the size of `m_achString` or add another member variable, you have to reissue your DLL. Unfortunately, there aren't any consistent ways to share C++ classes between applications.

That's where COM comes in. COM is designed to solve these exact problems. Let's start by looking at the difference between C++ objects and COM objects.

1. Whereas C++ objects are instantiated using operator new, COM objects are instantiated via an API function.
2. Whereas C++ objects are deleted using operator delete, COM objects are reference counted and self-destruct when they are no longer referenced by a client.
3. Whereas C++ uses specific mechanisms to perform run-time type information and typecasting, COM classes support run-time type information and typecasting using an established member function.
4. Whereas C++ classes can expose data as well as member functions, COM classes expose only member functions.
5. COM classes make no layout-specific assumptions.
6. COM introduces the notion of an interface to formalize the concept of an object pointer. More on this very shortly.

There's a lot there to swallow. In essence, COM does at the binary level most of what C++ does at the source code level. Essentially, using COM classes involves (1) making a request to the operating system to instantiate a COM object, (2) getting an interface pointer to the object, (3) using the interface, and (4) finally releasing the interface. If you're at all tuned into OLE, you've probably at least heard the term *interface*. The time of reckoning is here: What the heck is an interface?

Do You Speak I-Speak?

One of the most important aspects of COM is the notion of an interface. Strictly speaking, an interface is defined as a group of (semantically related) functions. Whenever functionality can be expressed by a group of functions, they are gathered together and called an interface. Technically, an interface is just a set of pure virtual functions waiting to be implemented.

Another good way to think of interfaces is as bundles of related functions. For example, a real OLE interface called `IDropTarget` (actually defined by Microsoft) includes such functions as `DragEnter()`, `DragOver()`, `DragLeave()`, and `Drop()`. Notice how each of these functions appears to be important for implementing the drop side of a drag-and-drop data transfer operation. Because the functions are related, they are part of a single interface.

COM interfaces stand apart from COM classes. COM classes expose interfaces as the means by which clients can communicate with them. COM interfaces represent a contract between a COM class and its client(s). Interfaces define the manner in which the client and the class communicate, and so are strongly typed. By using interfaces to define the contract, different classes may implement an interface differently yet be interchanged in binary form. As long as the behavior conforms to the interface definition, there is no problem. For example, imagine several classes that implement Automation (by implementing `IDispatch`). Underneath the hood, the two classes may choose to implement the `IDispatch` in completely different ways. However, as long as the interfaces behave as expected, the COM classes can be switched and replaced without consequence.

Each COM interface is unique. This leads to another important interface characteristic: interfaces are immutable. Once an interface is defined and published, it cannot be changed. A new version of an interface becomes an entirely new interface and is assigned a new identifier (more on that in a moment)—even if it was changed by simply adding a parameter to one of its functions. This means that a new interface will never conflict with an older interface.

An interface is the only way a client has to communicate with a COM class. The interface pointer hides all aspects of a class's internal implementation from the client. Adding data to COM interfaces would cause problems because adding data introduces compiler dependencies.

The final point about interfaces is that COM classes can implement multiple interfaces. That is, a COM class may provide more than one set of services. For example, applications implemented as compound document servers have to provide several interfaces, including `IPersistStorage`, `IOleObject`, `IDataObject`, `IOleInPlaceObject`, and `IOleInPlaceActiveObject` (never mind exactly what these interfaces do for now—they just have to be there for OLE documents to work). MFC relies on its document/view architecture to support OLE document technology. MFC implements these interfaces within the `COleServerDoc` class.

Notice that by convention all COM interfaces begin with the letter *I*. For example, there's `IDataObject`, which contains functions for performing Uniform Data Transfer. A COM interface called `IDispatch` handles OLE Automation. Why, there's even an interface called `ICaramba` that you can use when you're angry because your code won't compile (well, not yet—but there's no reason why there couldn't be an `ICaramba` interface).

Globally Unique Identifiers

Before moving ahead, let's talk briefly about globally unique identifiers (GUIDs), because they're used all over the place in COM.

Think about regular C++ classes for a moment. When you write a C++ class, you give it a name (like "class foo", or something equally creative). Then the member functions and member variables get names, too. The compiler uses this information to compile and link your application or library. These symbols are necessary for the compiler to resolve the variable addresses and function calls within the completed program. However, these names disappear once the executable or library is generated (except in the special case of exported functions). An object-oriented system in which clients written using one language can instantiate objects written in another language and that allows software to communicate at the binary level requires an identification system that persists even after all the compiling and linking is completed.

COM uses 128-bit numbers called globally unique identifiers to identify OLE classes (and interfaces, as you'll see later). A GUID looks something like this:

```
{00020900-0000-0000-C000-00000000046}
```

In fact, this GUID represents a Word for Windows document as an OLE document content object.

These numbers define OLE classes and interfaces throughout the system. They are so large (between 0 and 2^{128}) that collisions between these numbers are effectively prevented. In fact, Microsoft's COM specification states that the most common algorithm used to produce GUIDs is in theory "capable of allocating GUIDs at a rate of 10,000,000 per second per machine for the next 3240 years."

COM uses these numbers to uniquely identify COM classes and interfaces. By convention, COM class IDs use the prefix "CLSID" and interface IDs use the prefix "IID". For example, we'll work with a COM class called CoMath that is identified by CLSID_CoMath. A certain interface defined by Microsoft called IUnknown is identified by the symbol IID_IUnknown.

Exploring the Great IUnknown

Of all the interfaces defined, IUnknown is probably the most important one. It's the granddaddy of all interfaces, and COM requires that all COM classes and all COM interfaces implement IUnknown. You'll see in a moment how IUnknown more or

less bootstraps a COM object. You can find `IUnknown` in the OLE header files. Here's `IUnknown`'s definition:

```
struct IUnknown {
    virtual HRESULT QueryInterface(IID& iid, void **ppvObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

Note: Microsoft actually wraps these definitions using macros like `DECLARE_INTERFACE` and `STDMETHOD`, which take care of differences between 16- and 32-bit implementations. They're omitted here for the sake of clarity. Also, `HRESULT` is a rich error code that gives all sorts of information about the result of a function.

Notice that `IUnknown` is defined simply as a struct with three pure virtual functions. Any class that implements `IUnknown` has to implement these three functions. The functions support two areas of functionality required by all COM classes: object lifetime management and interface negotiation (or run-time discovery of functionality).

Object Lifetime Management

One important characteristic of COM objects is that they manage their own lifetimes. COM clients may hold several interface pointers to a single object multiple times (that is, COM objects may have more than one interface in use at a time). Another way of thinking about this is that COM objects may be referenced more than once simultaneously. COM objects need to keep themselves in memory for as long as they're still in use. This means that the COM object must manage its own lifetime and delete itself when it detects that it's no longer in use (that is, when there are no more outstanding interface pointers). COM objects self-destruct rather than depend on clients to delete them. To accomplish this, COM objects maintain reference counts on themselves (very much the way reference counts are maintained by DLLs). The COM object knows it's in use as long its reference count is greater than zero.

COM object reference counts are managed through `IUnknown`'s `AddRef()` and `Release()` functions. Whenever a client acquires an interface pointer to an object, the object's reference count is incremented by 1 through a call to `IUnknown`'s `AddRef()` function. When the client of the object is done with the interface, the client calls `Release()`, which decrements the object's reference count. Whenever the object is no longer in use (that is, its reference count drops to zero), the object can delete itself from memory.

To illustrate this, consider a simple COM object called `CoSomeObject`, which supports a single interface, `IUnknown` (of course). You would define the class as shown in Listing 11-3, using C++.

Listing 11-3. CoSomeObject: A simple COM class defined

```
class CoSomeObject : public IUnknown {
public:
    HRESULT QueryInterface(IID& iid, void FAR* FAR*ppvObj);
    ULONG AddRef();
    ULONG Release();

    CoSomeObject();

private:
    DWORD m_dwRefCount;
};
```

Naturally, you want to initialize the reference count to zero in the constructor. Other than that, implementing AddRef() and Release() is trivial. The AddRef() and Release() functions look something like Listing 11-4.

Listing 11-4. Implementation code for CoSomeObject

```
CoSomeObject::CoSomeObject() {
    m_dwRefCount = 0;
}

ULONG CoSomeObject::AddRef() {
    return ++m_dwRefCount;
}

ULONG CoSomeObject::Release() {
    DWORD dw;
    dw = m_dwRefCount--;
    if (m_dwRefCount == 0)
        delete this;

    return dw;
}
```

The code in Listings 11-3 and 11-4 is standard for implementing AddRef() and Release(). Notice that Release() deletes the object's this pointer. This may look a bit odd, but it's perfectly legal C++ syntax that does exactly what we want it to do—it lets the object control its own lifetime. Because only the object knows whether or not it's still in use, the object has to take responsibility for deleting itself.

The second issue handled by IUnknown is interface negotiation (also known as run-time discoverability). This is performed by the QueryInterface() function.

Interface Negotiation

To do anything useful, COM classes need to support more than one interface (remember, **IUnknown** is mandatory, so an OLE class should implement another interface like **IDataSource** or **IOleObject**). Once a client has a pointer to an object's interface, that client needs to be able to find out what other interfaces it might support. The way to do that in COM is to use an object's **QueryInterface()** function (which is supported through the **IUnknown** interface). **QueryInterface()** has this prototype:

```
HRESULT QueryInterface(REFIID riid, LPVOID far* ppvObj);
```

QueryInterface() takes two parameters: (1) a single, unique, 128-byte number identifying the interface to retrieve (you know—a GUID) and (2) a place to hold the interface pointer. When a client calls an interface's **QueryInterface()** function, the object compares the requested interface ID to each of the interfaces it supports. If **QueryInterface()** finds a match for the interface ID, it returns a pointer to the requested interface and increments the object's reference count using **AddRef()** on the way out. Because the object has just doled out an interface, it's officially in use, so it increments its reference count.

Now, imagine that **CoSomeObject** supports another interface called **IPersist**. (Incidentally, **IPersist** is a real interface defined by Microsoft.) Listings 11-5 and 11-6 show an implementation of **QueryInterface()** that returns a pointer to the interface requested in the **riid** parameter.

Listing 11-5. The definition of **CoSomeObject**

```
class CoSomeObject : public IUnknown, public IPersist {
    // IUnknown methods
    virtual DWORD AddRef(void);
    virtual DWORD Release(void);
    virtual HRESULT QueryInterface(REFIID,
                                  LPVOID FAR* );
    // IPersist methods
    virtual HRESULT GetClassID(LPCLSID pclsid);
};
```

Listing 11-6. The member functions of **CoSomeObject**

```
HRESULT
CoSomeObject::QueryInterface(IID& riid,
                            void FAR * FAR * ppvObj) {
    HRESULT hr = ResultFromSCode(E_NOINTERFACE);
    *ppvObj = NULL;
```

```

if (riid == IID_IUnknown) {
    *ppvObj = (IPersist *)this;
    hr = NOERROR;
} else if(riid == IID_IPersist) {
    *ppvObj = (IPersist *)this;
    hr = NOERROR;
}
return hr;
}

```

This is one way to implement `IUnknown::QueryInterface` (there are other ways, which you'll see later). Notice how this code goes about discovering interfaces. `QueryInterface()` checks the `riid` parameter against each of the object's available interfaces until one is found (or is determined to be unavailable). The class's `this` pointer is cast to `IPersist*` (remember, `IPersist` derives from `IUnknown`, so the `IUnknown` functions are available through that pointer) and the interface pointer is passed back in the `ppvObj` parameter. This implementation works for classes that use multiple inheritance to implement COM classes. As you'll see in a few moments, multiple inheritance is but one way to implement COM classes. Like `AddRef()` and `Release()`, this is just standard COM code for implementing interface lookup. Of course, this isn't the only way to implement interface lookup. You might use a lookup table or some other optimized method. In fact, MFC uses a lookup table similar to message maps to implement `QueryInterface()`.

A Peek at the Client Side: Call/Use/Release

`IUnknown`'s object lifetime control mechanism imposes a standard protocol for using COM interfaces from the client side. The call/use/release rules for using COM objects are as follows:

1. Call some function to acquire an interface (either by instantiating an object or calling `QueryInterface()` through an existing interface pointer).
2. Use the interface's functions as much as you like.
3. When you're done using the interface, call `Release()` through the interface pointer to decrement the object's reference count.

For example, OLE memory allocation happens through an interface called `IMalloc`. One way to retrieve an `IMalloc` pointer is by using a COM helper function called `CoGetMalloc()`. Having acquired a pointer to OLE's task memory allocator through `IMalloc`, you can call any of its functions, including `Alloc()` and `Free()`. Then when you're done with the interface pointer, you simply call `IMalloc`'s `Release()`

function. Because the object on the other end of the IMalloc pointer keeps a reference count, it'll delete itself from memory if necessary. Listing 11-7 illustrates the call/use/release protocol.

Listing 11-7. Example usage of the IMalloc interface

```
HRESULT AllocMem() {
    LPMALLOC pMalloc;
    LPSTR lpsz;

    ::CoGetMalloc(MEMCTX_TASK, &pMalloc);
    if (lpsz = LPSTR(pMalloc->Alloc(256)))
        result = NOERROR;
    else
        result = ResultFromScode(E_OUTOFMEMORY);

    if(lpsz)
        pMalloc->Free(lpsz);

    pMalloc->Release();

    return result;
}
```

Notice how the call/use/release protocol works: CoGetMalloc() retrieves an IMalloc pointer. IMalloc has several functions for managing memory, including Alloc(), Free(), Realloc(), and DidAlloc(). Use the functions as much as you like. Just make sure that you release the interface pointer when you're done. Notice that the interface pointer has to be released regardless of whether the functions succeeded or failed.

COM Object Servers

Naturally, COM classes have to live somewhere—either in a DLL or an EXE file. COM classes live in *servers*. COM servers come in two basic flavors: in-proc servers and out-of-proc servers. (There are also in-proc handlers but they are not relevant to this discussion.) In-proc servers are DLLs, and they inhabit the process space of their clients. Out-of-proc servers live their lives as EXE files and inhabit different process spaces (and are eventually even on different machines) from their clients.

In-Proc Servers (DLLs)

In-proc servers are implemented as DLLs under Windows. An in-proc server can be loaded into the client's process space and serves "in-process objects."

The main advantage of in-proc servers is speed: they are generally quite fast. Making an interface function call between an EXE and a DLL is very quick because the program and the DLL share the same process space. However, in-proc servers can be risky: if the DLL crashes, it can take the whole process space down with it.

Out-of-Proc Servers (EXEs)

An out-of-proc server runs in a separate process space (either on the same machine or on another machine). Out-of-proc servers existing on the same machine as the client are called "local servers," whereas servers existing on a different machine from the client are called "remote servers." Out-of-proc servers exist as EXE files.

EXE servers deliver a robustness that DLL servers cannot. If client code is calling the interface functions of an object in another process space, the client and the server are protected from one another. If the server crashes for some reason, the client process will remain unaffected. So, while there is the advantage of robustness, there is a downside to using out-of-proc servers: the cost of marshaling.

Here's the problem: the addresses in one process space (that is, data addresses and function pointers) make no sense in another process space. So how can it be that client code in one process space can call functions and pass parameters in another process space? The answer is through remote procedure calls (RPCs) and marshaling. To understand how all this works, let's dig into marshaling a little deeper.

All function calls and parameter passing must occur within the same process space—those are simply the rules of playing in the Windows game. However, COM's architecture has to somehow enable a client to communicate with any object through an interface pointer, either inside or outside of the client's address space. To accomplish this, OLE uses remote procedure calls. Here's how it works.

For each interface, a *proxy* is set up for the client code. A proxy is an in-process implementation of an interface. Because this code exists within the client's process space, the client code can call it easily—no fuss, no muss. However, the client isn't calling the actual server code (yet).

On the server end, a *stub* is set up for the specific interface (within the server's process space). A stub is the receiving end of the RPC transaction. It maps function calls and data sent by the proxy into function calls and data that make sense in the server's process space. When the client makes a call through the interface, the proxy packages the parameters into a portable 32-bit data structure. This may involve copying

buffers, strings, numbers, and other parameters. After the proxy packages the parameters together, it makes a remote procedure call to the server's process. On the server side, the stub maintains the real interface pointers, functions, and associated data structures. The stub unpackages the parameters so they make sense within its process space. When the function call returns, the stub packages the return values and any other parameters that return information and sends them back to the proxy. The proxy then unravels the return values and out parameters so the client can use them. The term *marshaling* refers to the actual process of packaging and unpackaging the parameters within different process spaces.

Figure 11-1 illustrates how the proxy/stub marshaling mechanism works.

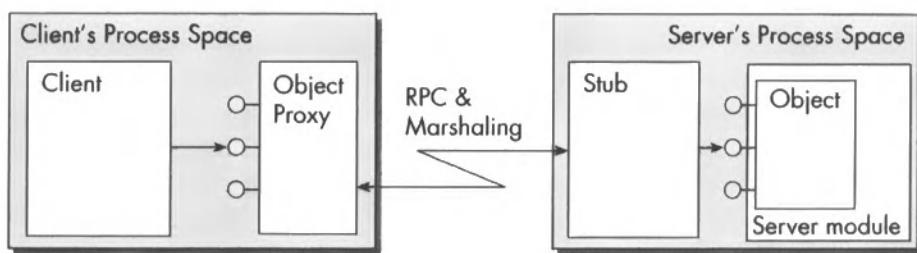


Figure 11-1. How COM objects communicate between process spaces

Class Factories

There is one more requirement for COM to be a truly binary standard: the server must allocate new objects on behalf of the client. Somebody's got to instantiate the COM object. Remember that in C++, new objects are allocated using operator new. Remember that COM moves software out into the binary realm. One of the main motivations for COM is to allow software of all different creeds and colors to intermingle. Operator new is a "C++-ism" and may not be present in all the languages people might want to use to build COM clients. For example, Visual Basic, the canonical OLE Automation client, knows nothing about allocating memory. To make COM work across all sorts of process spaces, object instantiation has to happen through an API function. And for that to work, the server must have an agent responsible for manufacturing the COM object at the behest of the client. Naturally, the agent is a COM object.

Consider the way you allocate objects in a C++ program. C++ objects are usually allocated by the client (most of the time by the actual program itself). A C++ program is then required to clean up after itself when it's done using the object. Listing 11-8 shows an example of this.

Listing 11-8. Object lifetime control in standard C++

```

class Foo {
    char m_szName;
public:
    Foo(char * szData);
    ShowData() {
        cout << m_szName << "\n";
    }
};

int main() {
    Foo *pFoo;

    pFoo = new Foo("This is the Foo Class");
    pFoo->ShowData();
    delete pFoo;

    return 0;
}

```

In effect, allocating and deallocating objects happens entirely within the domain of the program. Notice also how the code that created the object has direct access to all of the class's public member variables and functions.

By contrast, COM objects are always allocated by the server after a request from the client via a standard API function. When you think about it, there's really no other way. Remember, OLE is trying to achieve a binary standard. Because a client and a server may live in separate process spaces, there's no way a client can call operator new to create an object. But someone's gotta do it. That job falls to another COM class: the class factory.

COM enables this scenario through *class factories*. There is one class factory for every different *kind* of COM class within the server. Unfortunately, the term *class factory* may be a little misleading: class factories manufacture objects, not classes.

A class factory is simply another COM class. Like all other COM classes, class factories implement COM interfaces. In this case, the important interface is IClassFactory. IClassFactory consists of two functions: CreateInstance() and LockServer(). CreateInstance() is used by the client to create an instance of the OLE class. LockServer() increments a reference count on the entire server so that the server remains loaded in memory until all the clients are finished using it. Here's what the IClassFactory interface looks like:

```

struct IClassFactory {

    // IUnknown methods
    virtual HRESULT QueryInterface(IID& iid, void **ppvObj) = 0;

```

```

virtual ULONG AddRef() = 0;
virtual ULONG Release() = 0;

// IClassFactory methods
virtual HRESULT CreateInstance(IUnknown * pUnkOuter,
                               REFIID riid,
                               VOID ** ppvObject) = 0;
virtual HRESULT LockServer(BOOL fLock) = 0;
};

}

```

`IClassFactory` is no different from other COM interfaces. It's gotta implement `IUnknown` as well, so those function definitions are there too.

Again, every COM class gets its own class factory (at the source code level). For example, Figure 11-2 illustrates a COM server that holds a COM class called `CoSomeObject`. The COM server contains two COM classes: `CoSomeObject` and `CoSomeObject`'s class factory.

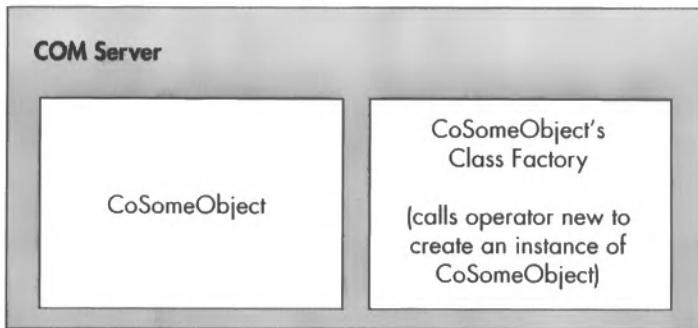


Figure 11-2. `CoSomeObject` and its class factory inside a COM server

Once the client requests the services of a COM class, how can it retrieve the class factory? In-proc servers accomplish this by exposing class factories through a well-known function, while out-of-proc servers register their class factories with Windows.

You might also note that the name `IClassFactory` is really not very appropriate. A toy factory produces toys. An `IClassFactory` does not produce classes (this the well-paid programmer does). An `IClassFactory` really creates objects, actual instances of a given class. Perhaps it should have been called `IOBJECTFactory`. This is probably why Microsoft calls its MFC class factory implementation `COleObjectFactory`.

Exposing the Class Factory in an In-Proc Server

Remember that in-proc servers are DLLs, and so can easily export functions. In-proc servers use exported functions to expose their class factories. That function is called `DllGetClassObject()`.

Clients can use one of two COM API functions to create instances of COM classes: `CoGetClassObject()` or `CoCreateInstance()`. Whenever a client calls one of these functions to create a COM object housed inside an in-proc server, COM calls the server's `DllGetClassObject()` function to get an interface pointer to the class factory for a particular COM class. Once the client has a pointer to the COM class's class factory, the client can call the class factory's `CreateInstance()` function to manufacture an instance of the class and obtain a pointer to one of its interfaces.

Here's the prototype for `DllGetClassObject()`:

```
STDAPI DllGetClassObject(REFCLSID rclsid,
                        REFIID riid,
                        LPVOID FAR*ppv)
```

This function takes three parameters: a reference to the GUID of the class to instantiate, the GUID representing the interface to acquire, and a place to put the interface pointer. The first parameter is easy. For example, if you want to retrieve `CoSomeObject`'s class factory, `CoSomeObject`'s GUID shows up in this parameter. The second parameter is usually `IID_IClassFactory`. However, if the class factory implements some other interface (besides `IClassFactory`), the client can request that interface ID specifically. The third parameter works the same way here as it does for `IUnknown::QueryInterface()`: it's a pointer to an interface pointer, a place to put the requested interface pointer.

You'll see exactly how `DllGetClassObject()` works in a real DLL shortly.

Exposing the Class Factory in an Out-of-Proc Server

Naturally, out-of-proc servers work a little differently from in-proc servers. You cannot just specify an exported function to retrieve the class factory. There needs to be some other mechanism. In fact, out-of-proc servers register their class factories at run time. OLE provides an API function expressly for this purpose. It's called (appropriately enough) `CoRegisterClassObject()`:

```
HRESULT CoRegisterClassObject(REFCLSID clsid,
                            IUnknown *pUnk,
```

```
    DWORD grfContext,
    DWORD grfFlags,
    LPDWORD pdwRegister);
```

Out-of-proc servers register their classes using `CoRegisterClassObject()`, usually during initialization. Notice that this function is a bit more complex than the exported DLL function. The first parameter is the GUID of the classes being registered. The second parameter is the object's `IUnknown` pointer. The third parameter is a set of flags indicating the context of the server—that is, whether the executable server code is a DLL, a local server, or a remote server. The context enumeration is a bitmap that can be bitwise ORed to include any of the server contexts:

```
enum tagCLSTX
{
    CLSTX_INPROC_SERVER      = 1,
    CLSTX_INPROC_HANDLER     = 2,
    CLSTX_LOCAL_SERVER       = 4,
    CLSTX_INPROC_SERVER16    = 8
} CLSTX;
```

Parameter four, a `DWORD` called `grfFlags`, indicates how connections are to be made to the server. This parameter can be any one of the values from the `REGCLS` enumeration, which determines the quantity of objects that can be created by a single class factory:

```
typedef enum tagREGCLS {
    REGCLS_SINGLEUSE = 0,
    REGCLS_MULTIPLEUSE = 1,
    REGCLS_MULTI_SEPARATE = 2
} REGCLS;
```

The last parameter is a pointer to a `DWORD` value that is handed out by COM when the class factory is registered. This value represents a token you can use to revoke the class factory's registration using `CoRevokeClassObject()`.

Once a class factory is registered with COM, clients can use the class factory to create instances of a class.

Unloading In-Proc Servers

You've seen that in COM, DLLs are loaded programmatically (usually through a call to `CoGetClassObject()` or `CoCreateInstance()`). So, how is the DLL unloaded?

Because a COM DLL is usually loaded programmatically, it has to be unloaded programmatically. If the DLL is never unloaded, it's doomed to stay in memory (*arrgh, no!!!*). However, because DLLs are passive, they have to be unloaded by an

outside agent. Fortunately, there's a COM API function called CoFreeUnusedLibraries() expressly for this purpose. COM clients should call this function periodically, usually at idle time. Working as a cleanup mechanism, CoFreeUnusedLibraries() checks each of the DLLs loaded by CoLoadLibrary() and asks each whether or not it can be unloaded. But how does it ask the DLL whether it can unload?

OLE in-proc servers provide a well-known function called DllCanUnloadNow(). COM calls a DLL's DllCanUnloadNow() function whenever the client calls CoFreeUnusedLibraries(). DllCanUnloadNow() returns S_FALSE under any of the following conditions:

1. There are any extant references to any of the DLL's class factory objects.
2. There are any extant references to any objects that the class factories may have created.
3. There are any outstanding calls to LockServer(TRUE) not reversed by LockServer(FALSE).

Otherwise, DllCanUnloadNow() returns S_OK. If the value returned by DllCanUnloadNow() is TRUE, COM unloads the DLL. Otherwise, COM lets the DLL remain in memory.

Unloading Out-of-Proc Servers

Like in-proc servers, out-of-proc servers need to maintain reference counts to indicate whether or not the server has been locked and whether or not there are any outstanding objects. The rules are the same as for in-proc servers, except that out-of-proc servers do not count extant references on their class factory objects as a reason for sticking around (that is, they obey all the preceding rules except number 1).

However, out-of-proc servers are different from in-proc servers because they are active. They don't wait for COM to unload them; they unload themselves. So, as clients use servers and the objects they provide, the server has to watch its own state as it doles out objects and interfaces. That means the server has to check its condition (1) whenever the client calls one of the class factory's LockServer() functions and (2) as clients release the interface pointers. The server can terminate itself under two conditions:

- When the client decrements the server's reference count (by calling IClassFactory::LockServer(FALSE) *and* there are no outstanding objects).
- When the client released the last object's final interface pointer *and* the server's lock count is zero.

Well, almost. There's one other stipulation for out-of-proc servers. Because they are whole executable programs in themselves, users can take control of the server whenever they want. For example, imagine opening Word for Windows and trying to link in an Excel spreadsheet. No problem: being a client, Word for Windows dutifully looks up Excel in the registry, fires it up, and uses the worksheet's class factory to create an instance of the worksheet object. Now imagine going over to Excel and opening another worksheet. Now Excel has two worksheets open: one connected to an OLE client and the other opened by the user. Had the user not opened up a new worksheet, Excel could terminate itself as soon as Word is terminated. However, in this case, the user has taken control, and it would be downright rude to shut down Excel while someone's working on a document.

Because of this, out-of-proc servers have to maintain one more piece of information about their state. That is, they have to maintain a flag indicating whether or not the user has taken control of the program (by opening another document, for example). That way, the server can either shut down gracefully (by warning the user, perhaps) or, even better, continue running until the user shuts down the server.

Class Registration in the Registry

Getting the class housed in a server is only half the battle. Next, clients need to know how to find the server. Windows keeps classes registered in the system registry. The system registry is a hierarchical database that maintains all sorts of information (OLE related and otherwise) about applications loaded on your computer. The registry has a list of GUIDs under the HKEY_CLASSES_ROOT section of the registry (see Figure 11-3). There's an entry for each server implementing the class represented by the GUID underneath each GUID's entry. For instance, if CoSomeObject were implemented in both an in-proc server and an out-of-proc server, the registry would contain two entries beneath the GUID's entry: "InprocServer32" and "LocalServer32". The values associated with these entries are the actual paths to each of the servers.

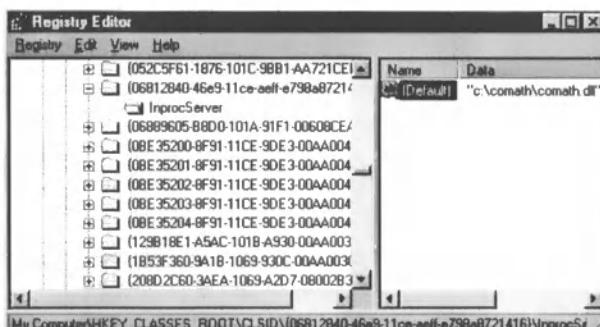


Figure 11-3. CoMath's InprocServer32 register entry viewed through REGEDT32

When an OLE client wants to instantiate an object, OLE looks up the GUID in the registry. If the request is for instantiating the object from within an out-of-proc server, OLE looks up the value “LocalServer32” to obtain the path to the EXE file that executes the server code. If the request is for instantiating an in-proc server, then the server uses the path associated with the “InprocServer32” key.

Creating Instances of COM Classes

COM clients use an OLE API function called `CoGetClassObject()` to retrieve a COM class’s class factory. Then, once the client has a pointer to the class’s `IClassFactory` interface, it calls `IClassFactory::CreateInstance()` to instantiate the COM object and obtain an interface pointer to it.

In addition, clients can call a more convenient function, called `CoCreateInstance()`, to instantiate COM classes. `CoCreateInstance()` performs several steps:

1. Looks up the server’s path in the system registry.
2. Loads the DLL or executes the EXE server.
3. Gets a pointer to the class’s class factory.
4. Creates an instance of the class using the `IClassFactory` interface.
5. Returns a pointer to the requested interface.

Figure 11-4 illustrates how OLE clients and in-proc servers work.

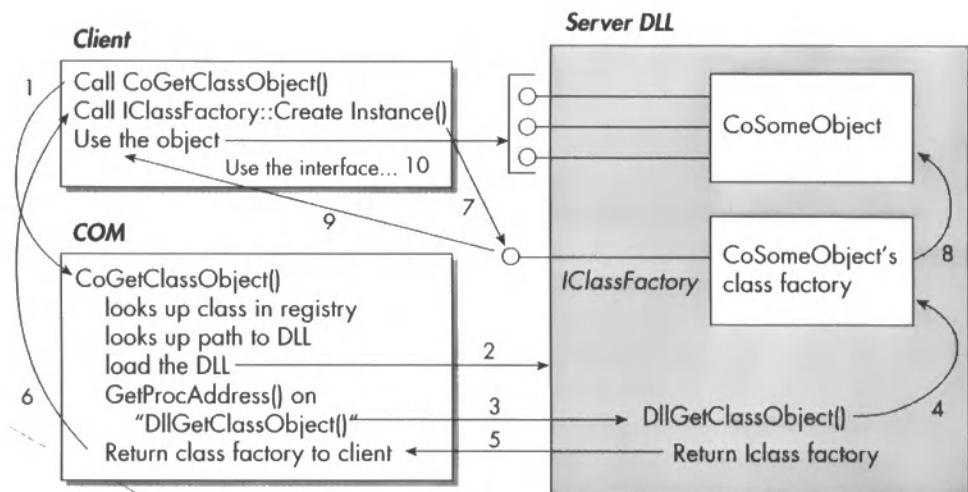


Figure 11-4. COM object creation using an in-proc server

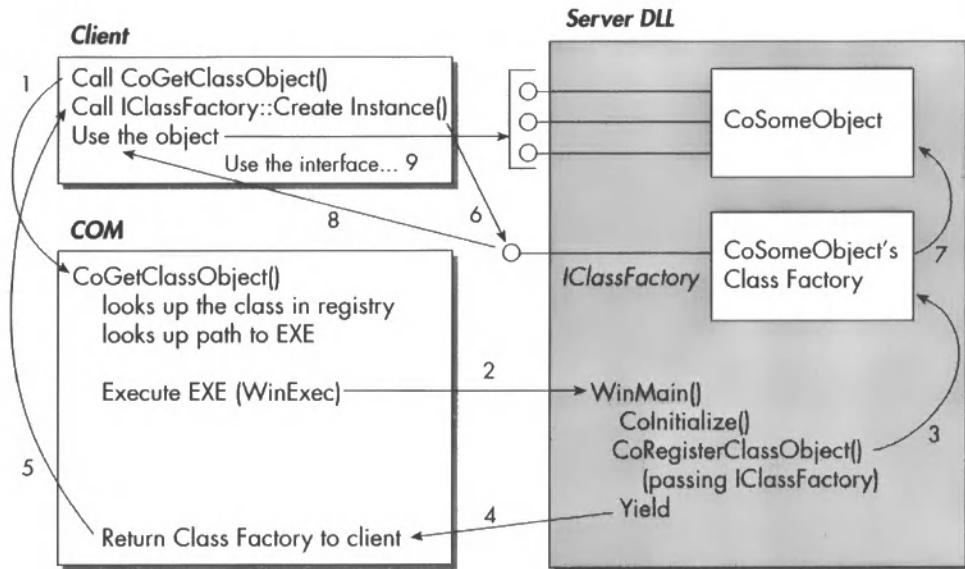


Figure 11-5. COM object creation using an out-of-proc server

Figure 11-5 illustrates how OLE clients and out-of-proc servers work.

In-proc and out-of-proc servers work in much the same way. The basic differences are in how they expose their class factories and how they are unloaded.

COM Classes with Multiple Interfaces

So, that's pretty much how COM works. Let's look at an example. To understand real COM classes with multiple interfaces, imagine a COM class called `CoMath`. `CoMath` will expose three interfaces: `IUnknown`, `IPersist`, and `IMath`. You've already seen `IUnknown`: that's the one interface required by all COM classes that deals with object lifetime and run-time discovery of other interfaces. To make use of that run-time discoverability, COM classes need to support some other functionality—other interfaces. This new COM class—`CoMath`—will also support `IPersist` and `IMath`. `IPersist` is already defined by Microsoft. It contains a single function called `GetClassID()`, which returns the GUID of the COM class implementing the interface. The only reason it's here is for illustration purposes. It's a really easy interface to implement (after all, it has only one function). The third interface implemented by this class is called `IMath`. `IMath` is a *custom interface*. That is, it's one that hasn't been defined by Microsoft (at least not yet). `IMath` is a simple interface with two

functions: Add() and Subtract(). Add() returns the sum of two numbers; Subtract() returns the difference between two numbers.

Listing 11-9 shows the actual IMath interface. Figure 11-6 is a graphical representation of the CoMath class.

Listing 11-9. The IMath interface

```
DECLARE_INTERFACE_(IMath, IUnknown)
{
    // *** IUnknown methods ***
    STDMETHOD(QueryInterface) (THIS_ REFIID riid,
                               LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG,AddRef) (THIS) PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;

    // *** IMath methods ***
    STDMETHOD(Add) (THIS_ INT, INT, LPLONG) PURE;
    STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG) PURE;
};
```

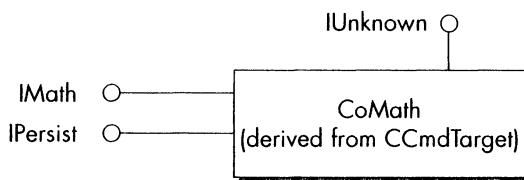


Figure 11-6. CoMath represented using Microsoft's plug-in jack notation

The two most common ways of implementing COM classes with multiple interfaces in C++ (at the source code level) are (1) using multiple inheritance and (2) using nested classes.

Implementing CoMath Using Multiple Inheritance

The first and most straightforward way to implement classes is through multiple inheritance. Simply define a C++ class that inherits from the interfaces you want to use. You can do that because interfaces are really just C++ structures that contain only pure virtual functions. Listings 11-10 and 11-11 show how to implement the CoMath class using multiple inheritance.

Listing 11-10. CoMath's header file (using multiple inheritance)

```
#ifndef _COMATH_H
#define _COMATH_H
#include "IMath.h"
```

```

class CoMath: public IPersist, IMath {
private:
    DWORD m_dwRefCount;

public:
    CoMath();
    virtual ~CoMath();

    // IUnknown methods
    STDMETHODIMP_(DWORD) AddRef(void);
    STDMETHODIMP_(DWORD) Release(void);
    STDMETHODIMP QueryInterface(REFIID riid,
                               LPVOID FAR* ppv);

    // IPersist methods
    STDMETHODIMP GetClassID(LPCLSID pclsid);

    // IMath methods
    STDMETHODIMP Add(INT x, INT y, LPLONG lpResult);
    STDMETHODIMP Subtract(INT x, INT y, LPLONG lpResult);
};

#endif // _COMATH_H

```

Listing 11-11. Implementing CoMath using multiple inheritance

```

CoMath::CoMath() {
    m_dwRefCount = 0;
}

CoMath::~CoMath() {
}

STDMETHODIMP_(DWORD) CoMath::AddRef(void) {
    return ++m_dwRefCount;
}

STDMETHODIMP_(DWORD) CoMath::Release(void) {
    DWORD dwResult = --m_dwRefCount;

    if(!dwResult)
        return dwResult;
}

STDMETHODIMP CoMath::QueryInterface(REFIID riid, LPVOID FAR *ppv) {
    *ppv = 0;
}

```

```

// Test for each interface here...
if (riid == IID_IUnknown)
    *ppv = LPPERSIST(this);
else if (riid == IID_IMath)
    *ppv = LPMATH(this);
else if (riid == IID_IPersist)
    *ppv = LPPERSIST(this);

// If the Interface was available, AddRef on the way out
if (*ppv) {
    LPUNKNOWN(*ppv)->AddRef();

    return NOERROR;
}

// Otherwise return the "No interface" error
return ResultFromScode(E_NOINTERFACE);
}

STDMETHODIMP CoMath::GetClassID(LPCLSID pclsid) {
    *pclsid = CLSID_CoMath;
    return NOERROR;
}

STDMETHODIMP CoMath::Add(INT n1, INT n2, LPLONG lpResult) {
    *lpResult = n1 + n2;
    return NOERROR;
}

STDMETHODIMP CoMath::Subtract(INT n1, INT n2, LPLONG lpResult) {
    *lpResult = n1 - n2;
    return NOERROR;
}

```

Using multiple inheritance to create COM classes makes a lot of sense. Notice how CoMath is derived from several different interfaces: IUnknown, IMath, and IPersist. CoMath inherits the member functions of these three interfaces. Remember, interface member functions are pure virtual. They are declared within IMath's definition and are implemented by CoMath.

Notice that CoMath maintains a member variable called dwRefCount. This is the object's reference count, used by the IUnknown functions to control object lifetime.

Implementing IUnknown is quite straightforward. AddRef() increments the object's reference count and Release() decrements it. QueryInterface() checks the requested interface against those supported by the object and returns the this pointer cast to the appropriate interface. The compiler takes care of returning the correct function table.

IPersist is the other interface defined by Microsoft. This one is even easier to implement: the interface's single function returns CoMath's GUID. Just put CoMath's GUID into the LPCLSID parameter and return NOERROR.

The other interface, IMath, has two functions that add and subtract a pair of numbers. Add() takes two integer parameters and a pointer to a LONG integer called lpResult to hold the result. Add() computes the sum and returns the result in lpResult. Subtract() is similar. It takes two integer parameters and a pointer to a LONG integer (also called lpResult) to hold the result. Subtract() computes the difference between the two integers and returns the result in lpResult.

All this action occurs within a single COM class that multiply inherits from IUnknown, IPersist, and IMath. In this case, the member functions for each interface are implemented by CoMath directly.

Implementing CoMath Using Nested Classes

The second method for writing a COM class that supports multiple interfaces is through class composition. This option is not quite as straightforward as the multiple inheritance option. However, it works just as well.

The idea behind using class composition is that there's a single class representing the COM class. This class in turn contains one or more nested classes that actually implement the interfaces.

Listings 11-12 and 11-13 illustrate an implementation of CoMath using nested classes.

Listing 11-12. CoMath's header file (using nested classes)

```
#ifndef _COMATHN_H
#define _COMATHN_H
#include "IMath.h"

class CoMath : public IUnknown {
private:
    DWORD m_dwRefCount;

public:
    CoMath();
    virtual ~CoMath();

    // IUnknown methods
    STDMETHODIMP_(DWORD) AddRef(void);
    STDMETHODIMP_(DWORD) Release(void);
    STDMETHODIMP QueryInterface(REFIID,
                               LPVOID FAR* );
}
```

```

class PersistObj: public IPersist {
public:
    CoMath* m_pParent;

    STDMETHODIMP_(DWORD) AddRef(void);
    STDMETHODIMP_(DWORD) Release(void);
    STDMETHODIMP QueryInterface(REFIID,
                               LPVOID FAR* );
    STDMETHODIMP GetClassID(LPCLSID pclsid);
} m_persistObj;

class MathObj: public IMath {
public:
    CoMath* m_pParent;

    STDMETHODIMP_(DWORD) AddRef(void);
    STDMETHODIMP_(DWORD) Release(void);
    STDMETHODIMP QueryInterface(REFIID,
                               LPVOID FAR* );

    STDMETHODIMP Add(INT, INT, LPLONG);
    STDMETHODIMP Subtract(INT, INT, LPLONG);
} m_mathObj;
};

#endif // _COMATHN_H

```

Listing 11-13. Implementing CoMath using nested classes

```

CoMath::CoMath() {
    m_dwRefCount = 0;
    m_persistObj.m_pParent = this;
    m_mathObj.m_pParent = this;
}

CoMath::~CoMath() {
}

DWORD CoMath::AddRef(void) {
    return ++m_dwRefCount;
}

DWORD CoMath::Release(void) {
    DWORD dwResult = --m_dwRefCount;
    if (!dwResult)
        delete this;
    return dwResult;
}

```

```
HRESULT CoMath::QueryInterface(REFIID iid, void** ppvObj) {
    if (iid == IID_IUnknown) {
        *ppvObj = (LPUNKNOWN)this;
        AddRef();
        return NOERROR;
    } else
    if (iid == IID_IPersist) {
        *ppvObj = (LPPERSIST)&m_persistObj;
        AddRef();
        return NOERROR;
    }
    else if (iid == IID_IMath) {
        *ppvObj = (IMath *)&m_mathObj;
        AddRef();
        return NOERROR;
    }
    return ResultFromScode(E_NOINTERFACE);
}

ULONG CoMath::PersistObj::AddRef()  {
    return m_pParent->AddRef();
}

ULONG CoMath::PersistObj::Release() {
    return m_pParent->Release();
}

HRESULT CoMath::PersistObj::QueryInterface(
    REFIID iid, void** ppvObj) {
    return m_pParent->QueryInterface(iid, ppvObj);
}

ULONG CoMath::MathObj::AddRef() {
    return m_pParent->AddRef();
}

ULONG CoMath::MathObj::Release() {
    return m_pParent->Release();
}
```

```

HRESULT CoMath::MathObj::QueryInterface(
    REFIID iid, void** ppvObj) {
    return m_pParent->QueryInterface(iid, ppvObj);
}

HRESULT CoMath::PersistObj::GetClassID(LPCLSID pclsid) {
    *pclsid = CLSID_CoMath;
    return NOERROR;
}

HRESULT CoMath::MathObj::Add(INT n1, INT n2, LPLONG lpResult) {
    *lpResult = n1 + n2;
    return NOERROR;
}

HRESULT CoMath::MathObj::Subtract(INT n1, INT n2, LPLONG lpResult) {

    *lpResult = n1 - n2;
    return NOERROR;
}

```

Figure 11-7 illustrates how CoMath and its contained classes are laid out.

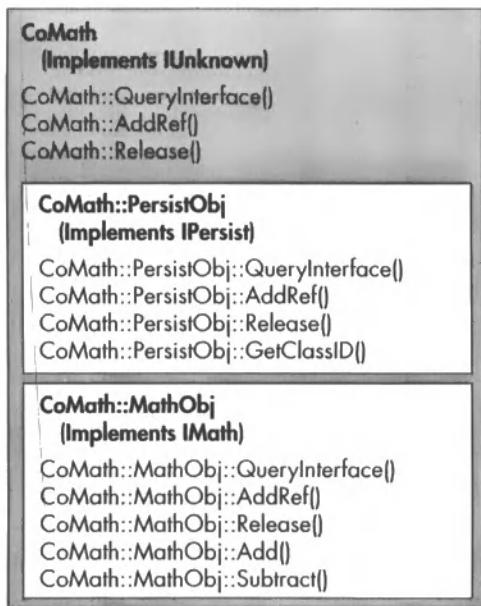


Figure 11-7. **PersistObj** and **MathObj** nested with **CoMath**

Notice that `IUnknown` is part of every interface—that is, every interface includes the three `IUnknown` functions: `AddRef()`, `Release()`, and `QueryInterface()`.

Think about `CoMath` for a moment. You want `IUnknown` to work on the entire class. That is, you want to keep a reference count on the whole class and you want to be able to query the whole class for its various interfaces. The composition version of `CoMath` does this by having the managing class—the class with object identity (`CoMath`)—implement `IUnknown`. Then each nested class implements `IUnknown` by delegating `AddRef()`, `Release()`, and `QueryInterface()` to `CoMath`'s implementation.

Again, this isn't the simplest way to implement classes with multiple interfaces, but it is fairly clear and works fine. In fact, this is very close to the way MFC implements COM classes.

A Class Factory for `CoMath`

Because `CoMath` is a COM class, it needs a class factory. Again, the class factory is another COM class that clients can use to manufacture instances of a specific COM class. As long as a specific class will be created from outside the server, that COM class needs a class factory. Listings 11-14 and 11-15 are a simple class factory for `CoMath`.

Listing 11-14. Header file for `CoMath`'s class factory

```
class CoMathClassFactory : public IClassFactory {

public:
    CoMathClassFactory();

    STDMETHODIMP_(DWORD) AddRef();
    STDMETHODIMP_(DWORD) Release();
    STDMETHODIMP QueryInterface(REFIID riid, LPVOID FAR *ppv);

    STDMETHODIMP CreateInstance(IUnknown * pUnkOuter,
                               REFIID riid,
                               VOID ** ppvObject);
    STDMETHODIMP LockServer(BOOL fLock);

private:
    DWORD m_dwRefCount;
};


```

Listing 11-15. Implementation of `CoMath`'s class factory

```
// global reference count for the server
static DWORD dwServerCount = 0;
```

```
static CoMathClassFactory coMathCF;

CoMathClassFactory::CoMathClassFactory() {
    m_dwRefCount = 0;
}

CoMathClassFactory::~CoMathClassFactory() {
}

DWORD CoMathClassFactory::AddRef(void) {
    return ++m_dwRefCount;
}

DWORD CoMathClassFactory::Release(void) {
    return --m_dwRefCount;
}

HRESULT CoMathClassFactory::QueryInterface(REFIID riid, LPVOID FAR *ppv)
{
    *ppv = 0;

    // Test for each interface here...
    if (riid == IID_IClassFactory)
        *ppv = LPCLASSTFACTORY(this);
    else if (riid == IID_IUnknown)
        *ppv = LPUNKNOWN(this);

    // If the Interface was available, AddRef on the way out
    if (*ppv) {
        LPUNKNOWN(*ppv)->AddRef();
        return NOERROR;
    }

    // Otherwise return the "No interface" error
    return ResultFromScode(E_NOINTERFACE);
}

HRESULT CoMathClassFactory::CreateInstance(IUnknown * pUnkOuter,
                                           REFIID riid,
                                           VOID ** ppvObject) {

    HRESULT hr = ResultFromScode(E_OUTOFMEMORY);

    CoMath *pCoMath = NULL;

    pCoMath = new CoMath;
```

```

if(pCoMath) {

    hr = pCoMath->QueryInterface(rIID, ppvObject);

    if(FAILED(hr))
        delete pCoMath;
}

return hr;
}

HRESULT CoMathClassFactory::LockServer(BOOL fLock) {
    if(fLock)
        dwServerCount++;
    else
        dwServerCount--;
};

}

```

The important thing to remember here is that the class factory represents the static area of a specific COM class. There's one instance of the class factory for every instance of a specific COM object.

Let's look at the class factory's `IUnknown` implementation. Because the server maintains an instance of `CoMathClassFactory` as a static variable, `Release()` doesn't delete the `this` pointer. The class factory exists for the life of the application. Also, `QueryInterface()` just examines the interface request and returns an interface pointer as long as the requested interface is `IID_IUnknown` or `IID_IClassFactory`.

Now let's look at the class factory's `IClassFactory` functions. `CoMathClassFactory::CreateInstance()` exists for the sole purpose of creating `CoMath` objects. Its mission in life is to allocate new `CoMath` objects on demand.

`CoMathClassFactory::LockServer()` manages a global variable within the server. Whenever `LockServer(TRUE)` is called, `LockServer()` increments the lock count. Whenever `LockServer(FALSE)` is called, `LockServer()` decrements the server's reference count.

The preceding code examples show how to implement COM classes using both multiple inheritance and nested classes. MFC takes the nested classes approach. Let's take a look at how MFC does it.

MFC COM Classes

One of the hurdles on the way to OLE enlightenment is the fact that OLE is in many ways just a specification about how to add various compelling features to your application. All those cool features like OLE documents, Automation, and controls

are implemented through the Component Object Model. To implement features like OLE documents and Automation, all you have to do is write the classes that implement the necessary interfaces. Of course, saying “you just have to implement the right interfaces” means one thing. Actually doing the work is another thing altogether.

For example, even though implementing Automation involves creating a class that supports a single interface called `IDispatch`, there’s nothing trivial about the interface itself. You have to implement all the operations to perform run-time lookup of functions, translate all the variant parameters, keep track of locality information, and several other issues. There are many ways of implementing `IDispatch` using a variety of languages and idioms. In fact, we’ll take a look at those methods in Chapter 14.

Automation is perfect work for a framework! (Enter MFC, stage left.) MFC provides a perfectly adequate implementation of `IDispatch` that’s quite easy to use (especially when you use the Class Wizard).

Another example is OLE document servers. An OLE document server has to implement a number of interfaces to qualify as an OLE document server. OLE document servers have to implement such interfaces as `IOleObject`, `IPersistStorage`, `IDataObject`, `IOleInPlaceObject`, and `IOleInPlaceActiveObject`. When you examine the OLE programmer’s reference, you’ll notice that some of these interfaces may contain upwards of 20 functions. All of a sudden you’re looking at a huge burden of work. (Forget about any dates this weekend!)

Again, this is the type of thing meant for a framework. Much of the effort involved in writing one OLE document server is the same no matter who’s doing it. That is, there’s a lot of boilerplate code.

The point is that underneath it all, MFC is just implementing a bunch of COM classes doing their stuff. It’s COM classes all the way down. The designers of MFC simply had to choose a way of implementing COM classes with multiple interfaces and start partying away on the code.

As it turns out, MFC takes the nested class/composition approach for implementing COM classes.

However, before examining the MFC approach to COM in detail, we’d like to say a few words about why Microsoft chose the nested class approach to implement COM classes.

Why MFC Uses Nested Classes

After examining the two main ways of developing COM classes, it may strike you that the nested class/composition method of implementing COM interfaces is rather verbose and may cause you to develop carpal tunnel syndrome from all that typing. Why would Microsoft use the nested class/composition approach when multiple inheritance seems so clean and succinct?

(continued)

This is actually a very common question. The answer boils down to the following issues:

1. Microsoft's position is that people should not have to understand multiple inheritance to use MFC. As a general rule, MFC tends to avoid using it.
2. There are actually some nasty problems that come up when trying to use multiple inheritance to implement COM interfaces.
- 2a. Interfaces that share names of functions are difficult to define. It is difficult to write different implementations of the same function for any interface that shares a function name with another interface. For example, this is the reason IDataObject names its advise function DAdvise instead of just Advise. IOleObject also has an advise function, and implementing IDataObject and IOleObject in a single interface would create a name collision of IDataObject and IOleObject if the advise functions didn't have different names.
- 2b. It is difficult to inherit the implementation of a given interface. You'd think that if you had an implementation of IUnknown, say it was called CUnknown, that you'd be able to inherit that implementation everywhere. Something like "COleClientItem : public CUnknown, IOleClientSite, IAdviseSink". The problem is that that isn't the way C++ works: you still need to write an implementation for the interfaces that inherit (indirectly) from IUnknown. This is done easily with delegation—but then so is the delegation used by MFC in its implementation of COM. Multiple inheritance doesn't solve this important problem (it could, but it doesn't—perhaps a bad decision on the part of the C++ designers . . .). Because it doesn't solve the problem at hand, using it isn't worth it.
3. Multiple inheritance introduces several inefficiencies.
- 3a. Microsoft's compiler (at least the 16-bit version) generates vtables that are unnecessary (one per interface, even if the specific functions in those interfaces were not overridden in derived classes).
- 3b. Inefficiencies inherent in multiple inheritance (this is hidden code generated by the compiler). For example: Did you know that the compiler has hidden NULL pointer checks in your examples?

```
*ppv = (IPersist*)this;
```

actually expands to:

```
if (this == NULL)
    *ppv = NULL;
else
    *ppv = this+IPersist_offset;
```

This hidden code bloats code unnecessarily, which is required by the definition of C++. It can be avoided if you know that "this" is never NULL. It may seem minor, but in a real program the extra code can add up significantly.

Overall it seems as though Microsoft felt there was not much advantage to using MI. In fact, as shown above, there are significant disadvantages. Given that much of the "grunge" can be hidden away in the MFC code, MFC macros, and perhaps in a Wizard (for the future), they chose the nested class/composition approach to implementing multiple interfaces.

Using MFC to Create CoMath

Now let's take a look at how MFC implements COM. To do so, we'll use MFC to create the same CoMath class we saw earlier. Then we'll take CoMath and put it inside a simple in-proc server. MFC's approach is basically the same as the nested-class approach just outlined. However, MFC hides much of the implementation behind the CCmdTarget class and something called interface maps. By examining this, we'll discover how MFC implements COM so that it doesn't seem so much like black magic.

IUnknown and CCmdTarget

MFC uses CCmdTarget (with the help of something called interface maps) to implement COM classes. In the nested-class example outlined previously, CoMath is the managing class that implements IUnknown. Notice how AddRef(), Release(), and QueryInterface() are all rolled by hand. In MFC, CCmdTarget serves as the base class for MFC's OLE-related classes. CCmdTarget implements IUnknown—the one interface that all COM classes must implement. CCmdTarget implements two sets of IUnknown functions. One set actually does all the IUnknown work (that is, the actual reference counting and interface lookup): InternalAddRef(), InternalRelease(), and InternalQueryInterface(). The other set is used to aid in COM aggregation: ExternalAddRef(), ExternalRelease(), and ExternalQueryInterface().

You'll find both sets of functions defined in the CCmdTarget definition:

```
class CCmdTarget: public CObject {
public:
    DWORD InternalQueryInterface(const void*, LPVOID* ppvObj);
    DWORD InternalAddRef();
    DWORD InternalRelease();

    DWORD ExternalQueryInterface(const void*, LPVOID* ppvObj);
    DWORD ExternalAddRef();
    DWORD ExternalRelease();
...
};
```

So, what is COM aggregation? COM aggregation is a way to reuse software at the binary level. It is worthwhile to understand aggregation because MFC has built-in support for it.

COM Aggregation

Binary Reuse

One of the main goals of the Component Object Model is to move objects from the source code realm to the binary realm. COM deals with code reuse at the binary level by exposing objects as a collection of interfaces in a way that can be understood by clients implemented in a wide variety of languages.

COM resolves certain software reuse issues by providing a standard way of sharing code at the binary level. However, COM doesn't support binary reuse via inheritance. Instead, one of COM's approaches to reuse is a technique called aggregation. (There's also a technique called containment, but that's a different story.)

COM Aggregation

COM aggregation is a way of taking several COM objects (each implementing a number of interfaces) and hooking them together so that the interfaces appear as though they are all implemented by a single object.

From the COM client's viewpoint, all COM objects appear the same. That is, they all expose some collection of interfaces. When reusing objects, you'd like to be able to combine several prewritten objects so they appear to be a single cohesive unit (the regular, normal way). COM aggregation is the way to do that.

How COM Aggregation Works

COM aggregation depends on a contract between two or more COM objects. The first COM object is the one that the client sees, called the controlling object. Then there are one or more participating objects. Both the controlling object and the participating objects have to perform a few special steps to make COM aggregation work.

Imagine that you're writing a COM object for an important human client called CoMath that implements three interfaces: IUnknown (of course), IAddSubtract, and IMultiplyDivide. Figure 11-8 illustrates the CoMath object. From the COM client's point of view, there is only one object implementing three interfaces.

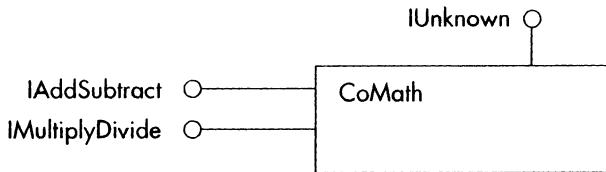


Figure 11-8. The client's view of the COM object

Now imagine that the CoMath object implements IAddSubtract just fine. However, your implementation of IMultiplyDivide is suboptimal, and your million-dollar client is demanding that you optimize it. Unfortunately, you've already signed your next

contract, and you don't have time to monkey around with your old source code. Fortunately, another software developer friend of yours has written a COM object that provides a superb implementation of IMultiplyDivide (perhaps it checks to make sure that a divide-by-zero never occurs). Naturally you'd like to use his algorithm, but your friend won't let you see the source code (some friend, huh?). In this case, what you want to do is license the binary version of your friend's object that implements IMultiplyDivide and integrate it into your own object. This is one example of a case in which you'd like to use COM aggregation.

Figure 11-9 shows two COM objects: CoMath and CoBetterMath (implemented by your friend).

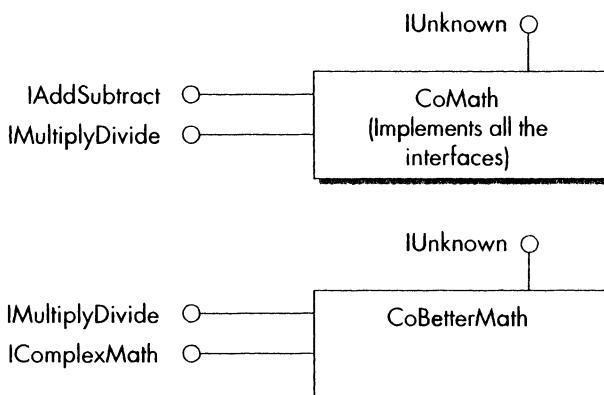


Figure 11-9. Two COM objects

CoBetterMath has an optimized implementation of IMultiplyDivide, which you'd like to use. However, the client wants to deal only with CoMath. How do you make the better IMultiplyDivide interface available as though it came directly from CoMath?

The key idea here is that even though there are several objects participating in the aggregate, the client must see the aggregate as a single unit. That is, the IUnknown functions (AddRef(), Release(), and QueryInterface()) should apply to the aggregate as a whole. For example, a QueryInterface() for IAddSubtract on your friend's IMultiplyDivide interface should return IAddSubtract even though your friend's IMultiplyDivide interface doesn't support IAddSubtract.

To get this to work, both objects must cooperate in the following way.

1. CoBetterMath (your friend's object) maintains a copy of your IUnknown interface pointer, called the controlling IUnknown. This is critical for COM aggregation to work.
2. Whenever an instance of your CoMath is created, it will in turn create an instance of CoBetterMath (the one that implements the better version of IMultiplyDivide) as part of the creation process. CoMath passes its IUnknown pointer to CoBetterMath.

3. When CoBetterMath receives the CoMath IUnknown pointer, it stores the CoMath IUnknown pointer. CoBetterMath now holds a controlling IUnknown pointer.
4. When the client asks CoMath for the IMultiplyDivide interface, CoMath simply passes out CoBetterMath's IMultiplyDivide interface.

So far, so good. Up to this point, everything seems to be working fine. Now the client has a copy of IMultiplyDivide that it can use to calculate all sorts of multiplication and division problems. But the client (which is holding an IMultiplyDivide pointer) encounters a situation where it needs the IAddSubtract interface. So the client uses QueryInterface() on IMultiplyDivide (remember now, this is CoBetterMath's IMultiplyDivide interface) asking for IAddSubtract. And what does the client get back? An error! Look at Figure 11-9 again. CoBetterMath doesn't implement IAddSubtract. There's still a piece missing from the arrangement.

CoBetterMath does have a connection to IAddSubtract: CoBetterMath has a copy of CoMath's IUnknown pointer. CoBetterMath uses this pointer by delegating all incoming IUnknown calls to this pointer. Figure 11-10 illustrates this relationship.

- 1) Client creates an instance of CoMath.
- 2) CoMath creates an instance of CoBetterMath, passing a copy of the CoMath IUnknown pointer
- 3) When CoBetterMath is created, CoBetterMath stores a copy of CoMath's IUnknown pointer
- 4) When the client asks CoMath to supply an IMultiplyDivide interface, CoMath responds by handing out CoBetterMath's IMultiplyDivide interface
- 5) CoBetterMath delegates all IUnknown calls to CoMath's IUnknown pointer, thereby preserving CoMath's IUnknown behaviour.

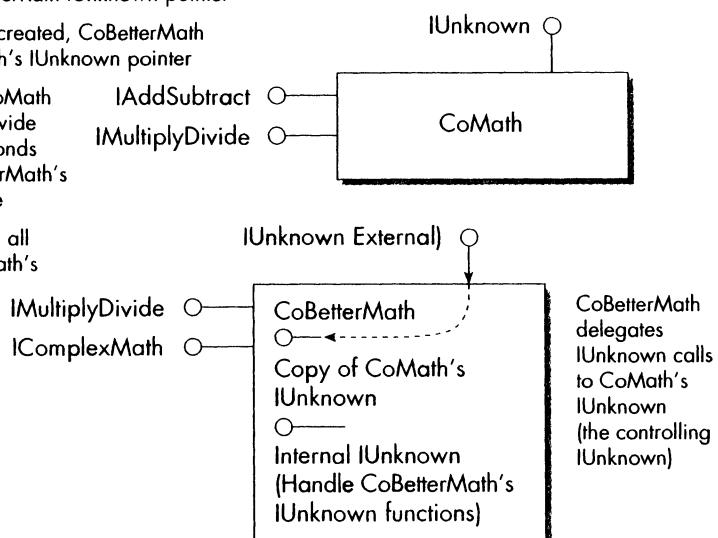


Figure 11-10. CoBetterMath using COM aggregation

To accomplish this, CoBetterMath implements two different sets of IUnknown functions: one set that delegates to the controlling IUnknown and one that manages CoBetterMath's regular IUnknown behavior. These two IUnknown pointers are usually

referred to as the delegating `IUnknown` and the nondelegating `IUnknown`, respectively. The delegating `IUnknown` forwards `IUnknown` calls to the controlling `IUnknown`, whereas the nondelegating `IUnknown` implements the object's regular reference counting and `QueryInterface()`. By default, the object participating in the aggregation forwards all incoming `IUnknown` calls to the controlling `IUnknown`. The aggregate knows whether or not it's participating in aggregation by the status of its controlling `IUnknown` (that is, whether or not it's `NULL`).

That's COM aggregation in a nutshell. It's a fairly efficient way of implementing binary reuse, and it doesn't require too much extra code. Aggregation is a key to COM programming and will become increasingly important in the future—especially when Distributed COM becomes available. Naturally, any application framework that implements COM should provide this facility. Fortunately, MFC does support aggregation within its COM implementation.

MFC and COM Aggregation

When you look through the MFC source code, you'll find two implementations of `IUnknown`: an internal version and an external version. These are the delegating and the nondelegating `IUnknowns`, respectively. `CCmdTarget` contains the following functions to implement `IUnknown`: `InternalQueryInterface()`, `InternalAddRef()`, `InternalRelease()`, `ExternalQueryInterface()`, `ExternalAddRef()`, and `ExternalRelease()`.

So, what's the difference between each version of these functions? Recall that to support aggregation, a COM object has to contain a pointer to a controlling `IUnknown`. If that `IUnknown` pointer is valid (that is, non-`NULL`), the object is participating in aggregation. In this case, the COM object forwards incoming `IUnknown` calls to the controlling `IUnknown`. That's exactly what `CCmdTarget`'s `External...()` `IUnknown` functions do. `CCmdTarget` has a data member called `m_pOuterUnknown` that is initialized to a controlling `IUnknown` whenever the object is created as part of an aggregate. If `m_pOuterUnknown` is valid (non-`NULL`), the `CCmdTarget`-derived class blindly calls the appropriate `m_pOuterUnknown` version of the function. For example, if the client calls `Release()` on the `CCmdTarget`-derived class participating in aggregation, the `CCmdTarget`'s `ExternalRelease()` calls `m_pOuterUnknown->Release()`. Otherwise, the `CCmdTarget`-derived class `InternalRelease()` simply decrements `CCmdTarget`'s reference count (`m_dwRef`). This is the standard way to implement COM aggregation.

There are two sides to COM aggregation: (1) making an object aggregatable and (2) being the controller of an aggregation. MFC provides support for both ends. Let's begin by looking at what it takes to make a COM class aggregatable.

Making a COM Class Aggregatable

Adding aggregation support to a `CCmdTarget`-derived class is fairly simple. `CCmdTarget` maintains an inner `IUnknown`, called `m_xInnerUnknown`, which handles the standard

IUnknown behavior for the object. EnableAggregation() turns on this inner unknown pointer, initializing it with an IUnknown vtable.

Using a COM Aggregate

Making a COM aggregate is a little more involved. It requires several steps:

1. Declare a member variable (an IUnknown*), which will contain a pointer to the aggregate object.
2. Include an INTERFACE_AGGREGATE macro entry in your interface map. This will refer to the member variable (the one pointing to the aggregate object) by name.
3. Initialize the member variable during CCmdTarget::OnCreateAggregates().

(Note: The macros used here may look a little odd. We'll cover MFC's interface maps in detail, starting on page 475.)

For example, Listing 11-16 shows how you might aggregate your friend's CoBetterMath object to get the better version of IMultiplyDivide.

Listing 11-16. COM aggregation using MFC

```
class CoMathAggr : public CCmdTarget {
public:
    CoMathAggr();

protected:
    IMultiplyDivide *m_lpMultiplyDivide;
    virtual BOOL OnCreateAggregates();

DECLARE_INTERFACE_MAP()

BEGIN_INTERFACE_PART(MathAggrObj, IAddSubtract)
    STDMETHOD(Add) (THIS_ INT, INT, LPLONG);
    STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG);
END_INTERFACE_PART(MathAggrObj)
};

CoMathAggr::CoMathAggr() {
    m_lpMultiplyDivide = NULL;
}

BOOL CoMathAggr::OnCreateAggregates() {
    CoCreateInstance(CLSID_CoBetterMath, GetConrollingUnknown(),
                    CLSCTX_ALL, IID_IMultiplyDivide,
                    (VOID FAR **)&m_lpMultiplyDivide);
```

```

if (m_lpTypeInfo == NULL)
    return FALSE;
}

BEGIN_INTERFACE_MAP(CoMathAggr, CCmdTarget)
    INTERFACE_PART(CoMathAggr, IID_IMath, MathObj)
    INTERFACE_AGGREGATE(CoMathAggr, m_lpMultiplyDivide)
END_INTERFACE_MAP()

```

What's Happening

INTERFACE_AGGREGATE inserts an IUnknown pointer representing a piece of the aggregate into the interface map. CCmdTarget has a function named QueryAggregates() that is called from within CCmdTarget::InternalQueryInterface(). This allows the aggregated interface pointers to be checked along with all the other interface pointers in the class's interface map.

COleObjectFactory's implementation of IClassFactory::CreateInstance() calls OnCreateAggregates() after creating the controlling object. This is the ideal place to create the other members of the aggregate.

Conclusion: COM Aggregation

OLE is above all an integration technology. One of the main goals of COM is to enable software reuse at the binary (as opposed to the source) level. COM aggregation is one way to do this. COM aggregation requires that objects participating in an aggregation maintain the controlling object's IUnknown pointer, to which they can delegate incoming IUnknown calls.

MFC supports COM aggregation, so CCmdTarget has two IUnknown implementations: InternalQueryInterface(), InternalAddRef(), and InternalRelease(); and ExternalQueryInterface(), ExternalAddRef(), and ExternalRelease().

Implementing COM aggregation in MFC is fairly straightforward. It requires using two new member functions (EnableAggregation() and OnCreateAggregates()) and a new macro (INTERFACE_AGGREGATE). Though a little hard to understand at first, COM aggregation is a wonderful way to reuse code once you've got the hang of it.

The Internal IUnknown Functions

To implement IUnknown's reference-counting functions (AddRef() and Release()), CCmdTarget keeps a reference counter, called m_dwRef. You'll find InternalAddRef() implemented in AFXOLE.INL. InternalAddRef() simply increments m_dwRef. InternalRelease() is a little more involved.

You'll find InternalRelease() implemented in OLEUNK.CPP. InternalRelease() is also a fairly standard implementation of IUnknown::Release(). It simply decrements the object's reference count and deletes the this pointer by calling CCmdTarget::OnFinalRelease(). CCmdTarget::OnFinalRelease() doesn't do much—it just executes "delete this;", which causes the COM object to delete itself from memory.

At this point you're probably asking, "What about QueryInterface()?" CCmdTarget::InternalQueryInterface() performs QueryInterface() using a lookup table called an interface map. We'll get to that in a few moments.

The External IUnknown Functions

You'll find ExternalAddRef(), ExternalRelease(), and ExternalQueryInterface() implemented in OLEUNK.CPP. CCmdTarget maintains a pointer to a controlling IUnknown interface (called m_pOuterUnknown) to support aggregation of the COM object if it was instantiated as part of an aggregate. All the external IUnknown functions work similarly. They first check to see if m_pOuterUnknown is valid. If there is a controlling IUnknown interface pointer, then External...() functions delegate the work to the controlling unknown. Otherwise, the External...() functions simply delegate the work to the CCmdTarget::Internal...() functions.

As in normal COM programming using C++, AddRef() and Release() aren't very much to talk about. They're almost trivial. The more interesting aspect of MFC COM classes is the way they implement IUnknown::QueryInterface() using interface maps.

Multiple Interfaces through Nested Classes

MFC supports multiple interfaces within a single class through class composition. MFC nests classes (each implementing a single interface) within the single outer class that supports the multiple interfaces. Recall the CoMath example that supports three interfaces: IUnknown, IMath, and IPersist.

You've already seen how IUnknown's object lifetime management functions (AddRef() and Release()) are implemented by CCmdTarget. And you know that CCmdTarget somehow implements IUnknown::QueryInterface(). However, it's not clear how MFC implements multiple interfaces and QueryInterface(). In the CoMath example, the IMath and IPersist interfaces are implemented by two nested classes: MathObj and PersistObj. Figure 11-11 illustrates the two nested classes contained within CoMath.

MFC uses macros to implement the nested classes and provide a lookup table for QueryInterface(). The next section describes interface maps and the macros that create them.

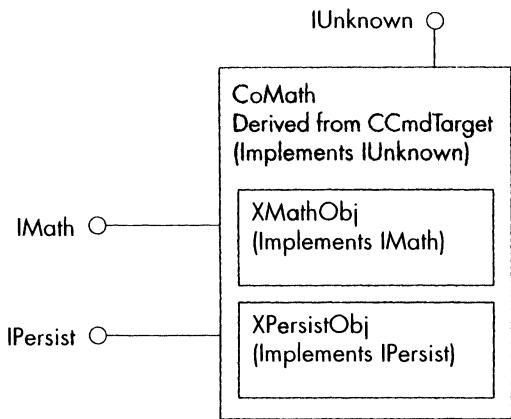


Figure 11-11. CoMath nests two separate classes

The MFC COM and Interface Map Macros

Remember that all COM interfaces must support `IUnknown`. That is, you want to be able to use `QueryInterface()` at any time to find out which other interfaces an object implements. That means that each of the nested classes within `CoMath` must also support `IUnknown`. Because the code for implementing `IUnknown` is virtually the same for all COM classes, MFC uses macros to factor out the redundant code. MFC uses several macros (working in conjunction with `CCmdTarget`) for doing this.

MFC's COM macros divide the work into two sections. Because MFC's implementation of COM classes is similar to the preceding nested-class example, part of the job of the macros is to set up the nested class structures. The second job of the interface map macros is to build the lookup table used for `QueryInterface()`.

Declaring the Nested Classes

MFC defines a pair of macros to declare the nested classes within the COM class, `BEGIN_INTERFACE_PART()` and `END_INTERFACE_PART()`, which you can find in the file `AFXDISP.H`.

Here's the `BEGIN_INTERFACE_PART()` macro:

```
#define BEGIN_INTERFACE_PART(localClass, baseClass) \
class X##localClass : public baseClass \
{ \
public: \
    STDMETHOD_(ULONG, AddRef)(); \
```

```
STDMETHOD_(ULONG, Release)(); \
STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj); \
```

This macro takes two parameters: the nested class implementing the interface and the interface itself. For example, to implement the IMath portion of CoMath, you would use this macro as follows:

```
BEGIN_INTERFACE_PART(MathObj, IMath)
```

Using this macro declares a nested class called XMathObj, derived from IMath. (MFC inserts the X to avoid polluting the name space.) BEGIN_INTERFACE_PART() also defines the three IUnknown functions for the COM class—AddRef(), Release(), and QueryInterface(). This is also where you declare all the member functions for the interface. So far, this is just like the nested class example from before.

The second macro for defining the nested classes is END_INTERFACE_PART(). This macro finishes the work started by BEGIN_INTERFACE_PART(). Here's the END_INTERFACE_PART() macro:

```
#define END_INTERFACE_PART(localClass) \
} m_x##localClass; \
friend class X##localClass; \
```

This macro simply closes off the declaration of the nested class and then declares a member variable of the nested class. Notice that END_INTERFACE_PART() declares that member as a friend of the managing class. This is done so that the nested class can access any private data members and functions of the managing class.

There's only one thing missing at this point: the other functions that belong to the interface. For example, IPersist has an extra member function (above and beyond IUnknown), called GetClassID(). The nested class hasn't implemented the interface until these member functions have been declared. Fortunately, this isn't difficult. All you have to do here is insert each of the function's prototypes. After all, what you're really doing is just declaring some member functions for the class. If you were to write the CoMath class using MFC (which we'll do momentarily), this is what you would see in CoMath's declaration within the header file:

```
// IMath members
BEGIN_INTERFACE_PART(MathObj, IMath)
    STDMETHOD(Add) (THIS_ INT, INT, LPLONG);
    STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG);
END_INTERFACE_PART(MathObj)

// IPersist members
BEGIN_INTERFACE_PART(PersistObj, IPersist)
```

```
STDMETHOD(GetClassID)(LPCLSID);
END_INTERFACE_PART(PersistObj)
```

So that's how the nested classes are implemented through macros. Let's take a look at how the interface map used to implement QueryInterface() is created.

Building the Interface Map

Like the other MFC maps (such as message maps and dispatch maps), interface maps correlate an identifier to some section of executable code. For example, message maps take a window message (such as WM_COMMAND) and map it to a member function within a C++ class (one derived from CCmdTarget, specifically). Interface maps work the same way. However, instead of mapping window messages to C++ class member functions, interface maps correlate an interface identifier (one of those 128-bit GUIDs) to one of the COM class's nested classes. That's the way MFC COM classes implement QueryInterface().

The interface map macros are similar to the macros used to implement other maps (such as message maps). For MFC's lookup tables, MFC usually provides a DECLARE...MAP() macro that goes in a class's header file and a BEGIN...MAP()/END...MAP() macro pair that goes in your class's C++ file. (Just substitute the ellipses with the specific map—like BEGIN_MESSAGE_MAP() or DECLARE_DISPATCH_MAP()). In keeping with its macro-based technology, MFC defines the following macros to implement interface maps: DECLARE_INTERFACE_MAP() and the BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP() pair. You'll find DECLARE_INTERFACE_MAP() in AFXWIN.H and the BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP() pair in AFXDISP.H.

DECLARE_INTERFACE_MAP()

Here's the DECLARE_INTERFACE_MAP() macro, as seen in AFXWIN.H:

```
#define DECLARE_INTERFACE_MAP() \
private: \
    static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[]; \
protected: \
    static AFX_DATA const AFX_INTERFACEMAP interfaceMap; \
    virtual const AFX_INTERFACEMAP* GetInterfaceMap() const; \
```

This macro defines the necessary machinery to insert an interface map into an MFC COM class. First, DECLARE_INTERFACE_MAP() declares an open-ended array of AFX_INTERFACEMAP_ENTRY structures. If you look inside AFXWIN.H, you'll see the AFX_INTERFACEMAP_ENTRY structure defined this way:

```
struct AFX_INTERFACEMAP_ENTRY
{
    const void* pid;
    size_t nOffset; // offset of the interface vtable from m_unknown
};
```

The first field contains a pointer to (what turns out to be) an interface ID. This is just a GUID assigned to identify a particular interface. The second field is an address. We'll see what that address is and how the field gets initialized shortly.

DECLARE_INTERFACE_MAP() also adds a member of type AFX_INTERFACEMAP. This is just a structure used to contain a pointer to the base class's interface map and a pointer to this class's interface map.

```
struct AFX_INTERFACEMAP
{
    const AFX_INTERFACEMAP* pBaseMap;
    const AFX_INTERFACEMAP_ENTRY* pEntry; // map for this class
};
```

Finally, DECLARE_INTERFACE_MAP() defines a function called GetInterfaceMap() that the framework can use to retrieve the object's interface map.

BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP()

Inserting DECLARE_INTERFACE_MAP() declares the interface map structures and functions for a particular COM class. Of course, now MFC has to insert some code in the CPP file. That's the job of BEGIN_INTERFACE_MAP() and END_INTERFACE_MAP().

The BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP() macros fill in the functions already defined by DECLARE_INTERFACE_MAP(). These macros are in AFXDISP.H. Here's BEGIN_INTERFACE_MAP:

```
#define BEGIN_INTERFACE_MAP(theClass, theBase) \
    const AFX_INTERFACEMAP* theClass::GetInterfaceMap() const \
{ return &theClass::interfaceMap; } \
const AFX_DATADEF AFX_INTERFACEMAP theClass::interfaceMap = \
{ &theBase::interfaceMap, &theClass::_interfaceEntries[0], }; \
const AFX_DATADEF AFX_INTERFACEMAP_ENTRY theClass::_interfaceEntries[] = \
{ }
```

BEGIN_INTERFACE_MAP() takes two arguments: (1) the class for which this interface map is being created and (2) that class's immediate ancestor. Inserting this macro in your source file yields the actual implementation of GetInterfaceMap(), which simply returns the class's interface map. This macro also initializes the class's

interfaceMap member variable to point to the first AFX_INTERFACEMAP_ENTRY in the table. Next, BEGIN_INTERFACE_MAP() begins filling the lookup table with AFX_MESSAGEMAP_ENTRY structures using the INTERFACE_PART() macro.

Remember that the AFX_MESSAGEMAP_ENTRY structure contains an interface ID (a GUID) and an offset into the CCmdTarget-derived class. The interface ID is that of the interface being implemented and the address is the offset of the nested class into the managing COM class. Here's INTERFACE_PART(), as it appears in AFXDISP.H:

```
#define INTERFACE_PART(theClass, iid, localClass) \
{ &iid, offsetof(theClass, m_x##localClass) }, \
```

MFC requires one INTERFACE_PART() entry for each interface being implemented in the COM class.

Finally, MFC defines the END_INTERFACE_MAP() macro to close off the interface map. This macro is also defined within AFXDISP.H:

```
#define END_INTERFACE_MAP() \
{ NULL, (size_t)-1 } \
```

Now that we've seen MFC's COM macros, let's look at how you might use them to build a real COM class.

The CoMath Class Using MFC

Now imagine using MFC to write the CoMath class: Remember that all real MFC COM classes derive from CCmdTarget. Again, that's because CCmdTarget implements IUnknown. Using MFC's interface maps, CoMath is defined in the header file as shown in Listing 11-17.

Listing 11-17. CoMath's header file (using MFC)

```
class CoMath : public CCmdTarget {  
  
protected:  
  
// IMath members  
BEGIN_INTERFACE_PART(MathObj, IMath)  
    STDMETHOD(Add) (THIS_ INT, INT, LPLONG);  
    STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG);  
END_INTERFACE_PART(MathObj)
```

```

// IPersist members
BEGIN_INTERFACE_PART(PersistObj, IPersist)
    STDMETHOD(GetClassID)(LPCLSID);
END_INTERFACE_PART(PersistObj)

DECLARE_INTERFACE_MAP()
DECLARE_OLECREATE(CoMath)

public:
// constructor/destructor
CoMath(void);
virtual ~CoMath(void);

protected:
DECLARE_DYNCREATE(CoMath)
};

```

Don't worry about the DECLARE_OLE_CREATE macro yet; that provides a class factory for CoMath. Otherwise, you've seen everything else.

Within the context of CoMath, BEGIN_INTERFACE_PART() does two things: (1) it declares nested classes implementing IMath and IPersist, and (2) it declares the IUnknown functions for that CoMath. END_INTERFACE_PART() closes off the nested class declaration and declares an instance of the nested class. Sandwiched between these macros are the interface functions above and beyond the IUnknown support for each interface. The DECLARE_INTERFACE_MAP() macro expands to declare an interface map for the class.

Listing 11-18 shows what the class looks like after the macros are expanded.

Listing 11-18. CoMath's header file after being preprocessed

```

class CoMath : public CCmdTarget {
protected:
    class XMathObj : public IMath {
public:
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();
    virtual HRESULT __stdcall QueryInterface(const IID & iid,
                                             LPVOID* ppvObj);
    virtual HRESULT __stdcall Add ( INT, INT, LPLONG );
    virtual HRESULT __stdcall Subtract ( INT, INT, LPLONG );
} m_xMathObj;
friend class XMathObj;

class XPersistObj : public IPersist {
public:
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

```

```

virtual HRESULT __stdcall QueryInterface(const IID & iid,
                                         LPVOID* ppvObj);
virtual HRESULT __stdcall GetClassID(LPCLSID);
} m_xPersistObj;
friend class XPersistObj;

private:
    static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[];
protected:
    static const AFX_INTERFACEMAP interfaceMap;
    virtual const AFX_INTERFACEMAP* GetInterfaceMap() const;
protected:
    static COleObjectFactory factory;
    static const GUID __cdecl guid;

public:
    CoMath(void);
    virtual ~CoMath(void);

protected:
public:
    static CRuntimeClass classCoMath;
    virtual CRuntimeClass* GetRuntimeClass() const;
    static void __stdcall Construct(void* p);

};

};

```

CoMath is now defined as a fully qualified COM class that supports three interfaces: IUnknown, IPersist, and IMath. The next step is to implement CoMath within the CPP file. That consists of two steps: (1) using the BEGIN_INTERFACE_MAP() / END_INTERFACE_MAP() macros to set up the QueryInterface() lookup table and (2) implementing each of the interfaces.

When you look at COMATH.CPP, you'll notice that that's where the interface map is actually implemented. CoMath implements two interfaces above and beyond IUnknown. Sandwiched between the BEGIN_INTERFACE_MAP() and the END_INTERFACE_MAP() macros are calls to the INTERFACE_PART() macro—one for each separate interface being implemented.

BEGIN_INTERFACE_MAP() implements the GetInterfaceMap() function for CoMath. Each INTERFACE_PART() macro adds an AFX_INTERFACEMAP_ENTRY (interface ID and class offset pair) to CoMath's interface map. CoMath has two classes, so it includes two interface map macros—one for each class. When the framework needs to access a class's interface, it gets a pointer to the interface map using GetInterfaceMap(). Then it walks the array of AFX_INTERFACEMAP_ENTRYs trying to find a matching interface ID. Here are the results of BEGIN_INTERFACE_MAP() and END_INTERFACE_MAP() after the preprocessor is finished:

```

const AFX_INTERFACEMAP* CoMath::GetInterfaceMap() const {
    return &CoMath::interfaceMap;
}

const AFX_INTERFACEMAP CoMath::interfaceMap = {
    &CCmdTarget::interfaceMap, &CoMath::_interfaceEntries[0],
};

const AFX_INTERFACEMAP_ENTRY CoMath::_interfaceEntries[] = {
    { &IID_IMath, (size_t)&((CoMath *)0)->m_xMathObj },
    { &IID_IPersist, (size_t)&((CoMath *)0)->m_xPersistObj },
    { 0, (size_t)-1 }
};

```

At this point, CoMath looks something like Figure 11-12.

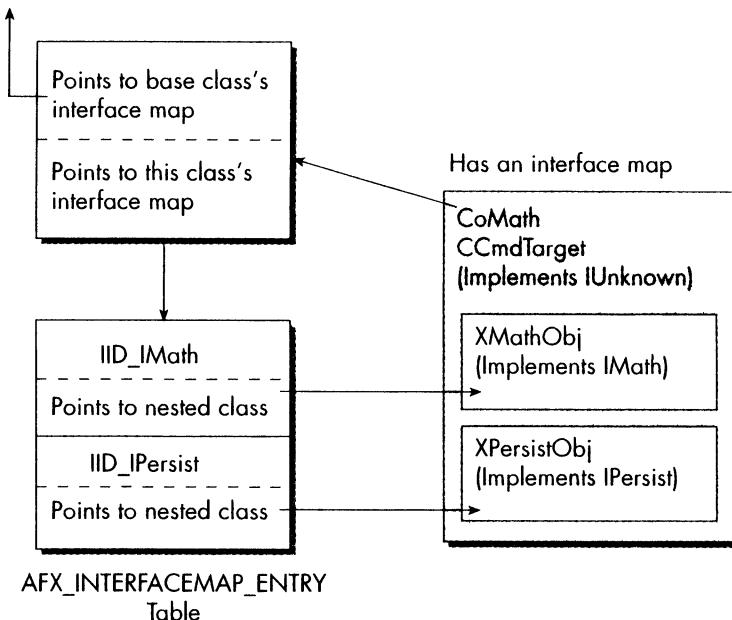


Figure 11-12. CoMath's interface map

MFC COM Classes and Inheritance

CoMath is an MFC class derived from CCmdTarget. It contains two nested classes, XMathObj and XPersistObj, each of which implements a COM interface. CoMath also carries an interface map that correlates an interface ID to a pointer to the nested class that implements the specific interface.

Figure 11-13 illustrates the hierarchy. Imagine an MFC class called CoMathEx that implements another interface called IMath2. Perhaps IMath2 implements two more functions called Multiply() and Divide(). You can use MFC to derive CoMathEx from CoMath, maintaining the functionality of CoMath's Add() and Subtract() functions.

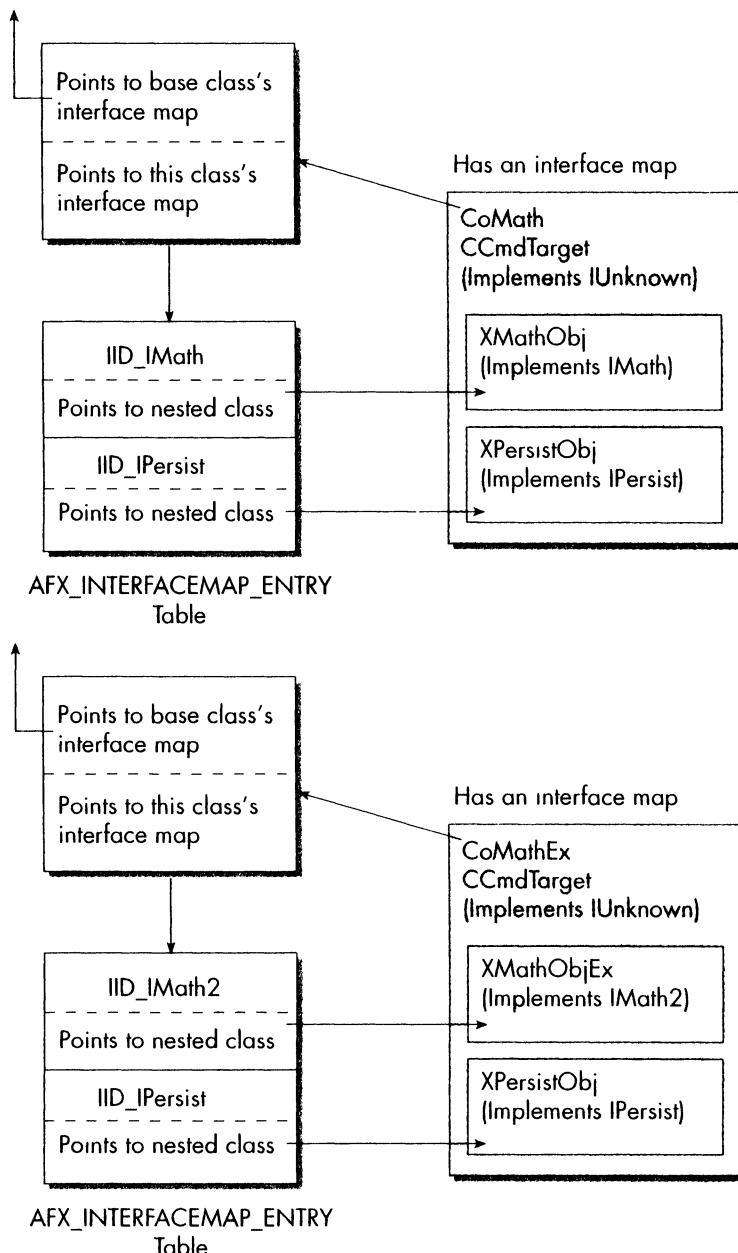


Figure 11-13. CoMathEx inherits CoMath's interface map

Notice how the interface maps work like message maps. If the interface isn't available for a specific class, MFC's interface map mechanism walks the list of all derived classes until it either finds the interface or determines that it's unavailable.

InternalQueryInterface() Revisited

So now the CoMath class has an interface map. What good does this do? Well, it turns out that CCmdTarget::InternalQueryInterface() uses the interface map to perform its duty. InternalQueryInterface() calls CCmdTarget::GetInterface() to retrieve the requested interface.

GetInterface() works like this. COM classes have an opportunity to install an interface hook. CCmdTarget has a virtual function to retrieve that interface hook. If GetInterfaceHook() returns TRUE, then GetInterface() is done. Otherwise, QueryInterface() still has some work to do. To perform QueryInterface(), GetInterface() simply walks the interface map looking for the interface ID in question. When it finds the interface, it returns the interface pointer (the address of the nested class) found in the AFX_MESSAGEMAP_ENTRY(), as shown in Listing 11-19.

Listing 11-19. Pseudocode for CCmdTarget::GetInterface()

```

LPUNKNOWN CCmdTarget::GetInterface(const void* iid) {
    Call CCmdTarget::GetInterfaceHook()
        to allow interface hook first chance
    If GetInterfaceHook returns a value,
        send that value back to the caller.

    Use CCmdTarget::GetInterfaceMap() to
        retrieve this object's interface map.

    Start walking the interface map towards
        the base class.

    do
        while there are still interface map entries {
            Try to match the interface ID in the map
            If match found,
                return the interface pointer
            else
                Get the next interface map entry
        }
        Get the previous next interface map (towards CCmdTarget)
    until there are not more interface maps
}

```

By the way, you can use GetInterface() whenever you want to work with a COM interface outside the scope of MFC's classes. For example, COleServerItem implements an interface called IDataObject. If you'd like to work with COleServerItem's IDataObject interface on its own terms, you can call GetInterface() to retrieve it. It takes the address of a GUID representing an interface (that is, an IID). If the MFC class implements that interface, then GetInterface() returns an IUnknown pointer. When the framework uses this function, it usually casts the IUnknown interface to the proper interface. This is safe to do because GetInterface() returns only if the requested interface is found. Here's the CCmdTarget::GetInterface() prototype:

```
LPUNKNOWN GetInterface(const void*);
```

Though GetInterface() and GetInterfaceHook() return pointers to interfaces, they do not AddRef() their return values; that is done at other layers, such as in InternalQueryInterface().

Finishing the Server

Once the interface maps have been defined, there are still a few more things left to do to finish the server. You need to (1) actually implement the interfaces (including the nested class's IUnknown functions), (2) assign a GUID to the CoMath class, (3) derive a class from CWinApp and declare an instance of it, (4) provide a class factory for the server, and (5) implement and export DllCanUnloadNow() and DllGetClassObject().

Implementing the Interfaces

By using interface maps in the CoMath class, CoMath uses nested classes for implementing IPersist and IMath. Even though the IUnknown functions were declared for each of these interfaces, you still need to provide an implementation of IUnknown for each of the subclasses. That is, you need to provide AddRef(), Release(), and QueryInterface() for CoMath::XPersistObj and for CoMath::XMathObj.

Remember, CoMath exposes two interfaces (above and beyond IUnknown), IPersist and IMath, and each of the classes implementing these two interfaces also needs to implement IUnknown. However, you want to be able to treat CoMath as a whole class. That is, you need to provide a single IUnknown implementation that works on behalf of the entire class. This is done by delegating each nested class's IUnknown implementation to CCmdTarget's IUnknown implementation.

You've seen CCmdTarget's IUnknown functions: ExternalAddRef(), ExternalRelease(), ExternalQueryInterface(), InternalAddRef(), InternalRelease(), and InternalQueryInterface().

To get a pointer to the COM object (in this case CoMath), you need to get a pointer to CoMath. Use the METHOD_PROLOGUE() macro, which takes two parameters: (1) the name of the outside class and (2) the name of the inner class implementing a specific interface. METHOD_PROLOGUE() declares a pointer to the outside class (CoMath), called pThis, and assigns the address of the outside controlling class to it. METHOD_PROLOGUE() does this by subtracting the offset of the inner class within CoMath. The pointer (pThis) can then access members of the controlling class (including ExternalAddRef(), ExternalRelease(), and ExternalQueryInterface()).

Use METHOD_PROLOGUE() anytime you need to get a pointer to the MFC COM class that implements your interfaces. For example, you would use METHOD_PROLOGUE() to implement each of the interfaces embodied by the nested classes within CoMath. Listing 11-20 shows how CoMath implements IUnknown for CoMath's IPersist interface:

Listing 11-20. Implementing IPersist's AddRef(), Release(), and QueryInterface() functions

```
STDMETHODIMP_(ULONG)
CoMath::XPersistObj::AddRef(void) {
    METHOD_PROLOGUE(CoMath, PersistObj)
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG)
CoMath::XPersistObj::Release(void) {
    METHOD_PROLOGUE(CoMath, PersistObj)
    return pThis->ExternalRelease();
}

STDMETHODIMP
CoMath::XPersistObj::QueryInterface(REFIID riid, LPVOID FAR*ppv) {
    METHOD_PROLOGUE(CoMath, PersistObj)
    return pThis->ExternalQueryInterface(&riid, ppv);
}
```

These functions also have to be implemented for XMathObj as well. Unfortunately there's no Interface Wizard for all this stuff yet: Microsoft will almost certainly provide one in the future. During the development of MFC 2.5, Microsoft had all this stuff wrapped up into macros. One issue that arose was that wrapping a bunch of code into a macro made it difficult to debug. It's very difficult to set a breakpoint in QueryInterface() if it is generated by one line of code that in turn generates other somewhat unrelated functions (that is, AddRef()/Release()). In the end, Microsoft removed the more complex macros to ease debugging and understanding. It's difficult

to write COM classes using MFC right now. However, in the future, the wizard will simplify doing COM classes, and you'll still be able to debug things.

Here's the definition of METHOD_PROLOGUE(), as found in AFXDISP.H:

```
#define METHOD_PROLOGUE(theClass, localClass) \
theClass* pThis = \
    ((theClass*)((BYTE*)this - offsetof(theClass, m_x##localClass))); \
```

METHOD_PROLOGUE takes two parameters: the COM class (derived from CCmdTarget) and the nested class. METHOD_PROLOGUE then gets the offset of the nested class into the COM class. For example, to get a pointer to CoMath from within m_xPersistObj, METHOD_PROLOGUE generates this code.

MFC Support for Class Factories

CoMath's not quite done yet. We'll revisit it in a minute. There's one more important COM aspect to cover first: class factories.

Recall that OLE class factories are just COM-compliant classes that implement the IClassFactory interface. Strictly speaking, all a class factory is supposed to do is implement IClassFactory. However, MFC includes several handy registration functions as part of their classes that implement IClassFactory. And why not? MFC's class factory implementation has the necessary GUID information. From there it's only a short step to registering class factories at run time. You'll find MFC's basic class factory support within COleObjectFactory.

COleObjectFactory

COleObjectFactory is a fully qualified COM class that implements IClassFactory. COleObjectFactory implements the interface in the MFC way—using interface maps. Inside the definition for COleObjectFactory, as shown in Listing 11-21, from AFXDISP.H, you'll find an interface map for IClassFactory.

Listing 11-21. Definition of COleObjectFactory

```
class COleObjectFactory : public CCmdTarget
{
    DECLARE_DYNAMIC(COLEObjectFactory)

    // Construction
public:
```

```

ColeObjectFactory(REFCLSID clsid, CRuntimeClass* pRuntimeClass,
    BOOL bMultiInstance, LPCTSTR lpszProgID);

// Attributes
BOOL IsRegistered() const;
REFCLSID GetClassID() const;

// Operations
BOOL Register();
void Revoke();
void UpdateRegistry(LPCTSTR lpszProgID = NULL);
    // default uses m_lpszProgID if not NULL
BOOL IsLicenseValid();

static BOOL PASCAL RegisterAll();
static void PASCAL RevokeAll();
static BOOL PASCAL UpdateRegistryAll(BOOL bRegister = TRUE);

// Overridables
protected:
    virtual CCmdTarget* OnCreateObject();
    virtual BOOL UpdateRegistry(BOOL bRegister);
    virtual BOOL VerifyUserLicense();
    virtual BOOL GetLicenseKey(DWORD dwReserved, BSTR* pbstrKey);
    virtual BOOL VerifyLicenseKey(BSTR bstrKey);

// Implementation
public:
    virtual ~ColeObjectFactory();
#ifdef _DEBUG
    void AssertValid() const;
    void Dump(CDumpContext& dc) const;
#endif

public:
    ColeObjectFactory* m_pNextFactory; // list of factories maintained

protected:
    DWORD m_dwRegister;           // registry identifier
    CLSID m_clsid;               // registered class ID
    CRuntimeClass* m_pRuntimeClass; // runtime class of CCmdTarget
                                    // derivative
    BOOL m_bMultiInstance;        // multiple instance?
    LPCTSTR m_lpszProgID;         // human readable class ID
    BYTE m_bLicenseChecked;
    BYTE m_bLicenseValid;
    BYTE m_bRegistered;          // is currently registered w/ system
    BYTE m_bReserved;             // reserved for future use

```

```

// Interface Maps
public:
    BEGIN_INTERFACE_PART(ClassFactory, IClassFactory2)
        INIT_INTERFACE_PART(COleObjectFactory, ClassFactory)
        STDMETHOD(CreateInstance)(LPUNKNOWN, REFIID, LPVOID*);
        STDMETHOD(LockServer)(BOOL);
        STDMETHOD(GetLicInfo)(LPLICINFO);
        STDMETHOD(RequestLicKey)(DWORD, BSTR* );
        STDMETHOD(CreateInstanceLic)(LPUNKNOWN, LPUNKNOWN, REFIID, BSTR,
            LPVOID* );
    END_INTERFACE_PART(ClassFactory)

    DECLARE_INTERFACE_MAP()

    friend SCODE AFXAPI AfxDllGetClassObject(REFCLSID, REFIID, LPVOID* );
    friend SCODE STDAPICALLTYPE DllGetClassObject(REFCLSID, REFIID, LPVOID* );
};

}

```

Version Note

Beginning with MFC version 4.0, Microsoft uses COleObjectFactory to implement IClassFactory2. In earlier versions, MFC used COleObjectFactory to implement IClassFactory and COleObjectFactoryEx to implement IClassFactory2. IClassFactory2 does everything IClassFactory does, as well as introduces support for component licensing.

COleObjectFactory's constructor takes four arguments: (1) the GUID of the COM class, (2) a pointer to the CRuntimeClass associated with the COM class, (3) a flag indicating whether or not a single instance of the server can create multiple instances of the COM class, and (4) a human-readable name for the server. When a COleObjectFactory is constructed, it stores the class factory's GUID, the CRuntimeClass of the COM class, the multiple-instance flag, and the program ID in its own member variables. It also sets dwRegister to zero to indicate that the class factory is not yet registered.

Because COM servers may house multiple classes, COM servers need to maintain multiple class factories. Remember that there is typically one class factory for each COM class inside the server. MFC keeps a list of class factories. The first node in this list is maintained in MFC's state information. Each time a class factory is constructed, it appends itself to the end of the application's list of class factories.

Class Factory and Server Registration

COleObjectFactory includes several functions for registering the class factory with OLE (for out-of-proc servers) and for updating the system registry. The first three pertain to a single instance of COleObjectFactory:

```

BOOL Register();
void Revoke();
void UpdateRegistry(LPCTSTR lpszProgID = NULL);

```

`COleObjectFactory::Register()` registers the class factory with OLE so that clients can use its services. There's not much to this function: it's really just a wrapper for `CoRegisterClassObject()`. Calling `CoRegisterClassObject()` registers the class factory with OLE. In doing so, OLE fills `m_dwRegister` with a token that `COleObjectFactory` can use later to revoke the class factory's registration.

`COleObjectFactory` has a companion function for `Register()`, called `Revoke()`. `Revoke()` cancels a class factory's registration with Windows. Similar to `COleObjectFactory::Register()`, `Revoke()` wraps the OLE API function `CoRevokeClassObject()` using the token that is represented by `COleObjectFactory::m_dwRegister`.

Finally, `COleObjectFactory::UpdateRegistry()` updates the system registry. `UpdateRegistry()` takes a single parameter: the prog ID of the class being registered. `COleObjectFactory::UpdateRegistry()` calls the AFX helper functions `AfxOleRegisterServerClass()` and `AfxOleRegisterHelper()` to perform the actual registration. `AfxOleRegisterServerClass()` and `AfxOleRegisterHelper()` are simply wrappers for several OLE-registry API functions.

The registration database is the central clearinghouse for information about COM classes. There are quite a few different kinds of items contained in the database. Consequently, registering all the items necessary for a complex program can become tedious.

Here are only a few of the many items that can appear in the registration database:

- **CISID**—The GUID representing the COM class.
- **InprocServer32**—The path to the in-proc server.
- **LocalServer32**—The path to the local server.
- **ProgID**—A human-readable string representing the COM class.
- **Verb**—An OLE document server verb.

`COleObjectFactory`'s `UpdateRegistry()` function adds the appropriate entries to the system registry.

The second three `COleObjectFactory` members are static member functions that pertain to all instances of `COleObjectFactory` in a specific application:

```

static BOOL PASCAL RegisterAll();
static void PASCAL RevokeAll();
static void PASCAL UpdateRegistryAll();

```

They perform the same functions as the nonstatic members, with the exception that these functions apply to all class factories within a single application. Each function walks the lists of class factories within the application, registering, revoking the registration, or updating the registry for each class factory.

Developer Tip: Registering Other Information

If you've ever tried to add a key to the registry, you know it can be a bit cumbersome. For example, if you want to add a custom verb to an OLE document server, you could do it by hand through the registry editor, but that's quite a dangerous proposition. I've seen developers inadvertently destroy a Windows NT installation by selecting the wrong menu items (we've even done it ourselves!). You could also register a custom verb using the registration API. Registering things programmatically requires several steps:

1. Open the CLSID key under HKEY_CLASSES_ROOT using RegOpenKey().
2. Replace the value using RegSetValue().
3. Close the key using RegCloseKey().

Along the way, you'll need to check error codes and handle special cases. For example, what if a value already exists for a particular key?

Of course MFC makes entries in the registry on an application's behalf. Why can't you? If you'd like, you can use AfxOleRegisterHelper() to update the registry. Here's the prototype for AfxOleRegisterHelper():

```
BOOL AFXAPI AfxOleRegisterHelper(LPCTSTR const* rgpszRegister,
                                LPCTSTR const* rgpszSymbols, int nSymbols, BOOL bReplace,
                                HKEY hKeyRoot = ((HKEY)0x80000000)); // HKEY_CLASSES_ROOT
```

AfxOleRegisterHelper() takes five parameters:

1. rgpszRegister is an array of strings containing both the keys and their values to add to the registry.
2. rgpszSymbols is an array of strings holding any substitutions you'd like to make. For example, you may want to substitute the class ID.
3. nSymbols indicates the number of symbols in the lpszSymbols array.
4. bReplace indicates whether or not to replace an entry in the registry.
5. bKeyRoot represents the root key for all these items. Because you want to register stuff for OLE, the default value is HKEY_CLASSES_ROOT.

So, how could you use this information? Imagine that you want to add a custom verb to your OLE document server. Imagine writing a compound document server that plays some sort of media clip (like the Microsoft Media Player). Naturally, one verb you want to add is a “Play” verb. MFC registers only “Edit” and “Open”. How can you add a “Play” verb?

Listing 11-22 shows the code that’ll do the trick. The best place to put this code is in your application’s initialization code. For example, CWinApp::InitInstance() is a great place to include this code.

Listing 11-22. Registering the custom verb “play”

```
{
...
//Add custom verb.
// Setup the verb string
static TCHAR szCustomVerb[] = _T("CLSID\\%1\\Verb\\2\0") _T("&Play,0,2");
LPCTSTR lpszRegister[2];
lpszRegister[0] = szCustomVerb;
lpszRegister[1] = NULL;

LPCTSTR lpszSymbols[2];
LPTSTR lpszClassID;
::StringFromCLSID(clsid, &lpszClassID);
lpszSymbols[0] = lpszClassID;
lpszSymbols[1] = NULL;

AfxOleRegisterHelper(lpszRegister, lpszSymbols, 1, TRUE);

AfxFreeTaskMem(lpszClassID);
...
}
```

Remember, AfxOleRegisterHelper() takes two arrays of pointers to TEXT strings. The first array contains the keys to register and their associated values. AfxOleRegisterHelper() expects these to be separated by a NULL character. In the preceding example, szCustomVerb contains the key in the first part of the string and the value in the second part. The first element of the lpszRegister array is the address of szCustomVerb. The second element is a NULL pointer. That’s how AfxOleRegisterHelper() determines the end of the list of registry entries.

The second array of strings is a set of symbols for substitution. In this case there’s only one symbol: the class ID of the object being registered. Again, this array is terminated by a NULL pointer. Notice the “%1” inside the szCustomVerb string. AfxOleRegisterHelper() uses that token to substitute the first symbol from the symbol array. If the string had contained “%2”, AfxOleRegisterHelper() would substitute the second element in the symbol array.

The first element of the symbol array is filled with the address of a string representing the object's class ID. StringFromClassID() is an OLE function that takes one of those unruly GUIDs and turns it into a nice, human-readable string.

The third parameter indicates that there is only one symbol in the symbol array. The fourth parameter tells AfxOleRegisterHelper() to overwrite the key and value if they're already registered.

By calling AfxOleRegisterHelper() this way, AfxOleRegisterHelper() adds this value: "&Play,0,2", for this key: "CLSID\{81634F00-DF4F-11CE-AEFF-A1970E160F92}\Verb\2". (Of course, your GUID will be different.)

The Heart of COleObjectFactory: OnCreateObject()

Because COleObjectFactory supports the IClassFactory interface, it must implement IClassFactory::CreateInstance(). COleObjectFactory::OnCreateObject() is where the magic occurs. Notice in the CoMath example that CoMath's class factory uses operator new to create an instance of CoMath. Recall that one of the parameters passed into COleObjectFactory's constructor is a pointer to the class's run-time class information. As long as that class implements dynamic creation (using the DECLARE_DYNCREATE()/IMPLEMENT_DYNCREATE() macros), the class's run-time information structure has a CreateObject() function that creates an instance of the class out of thin air. COleObjectFactory uses MFC's dynamic creation mechanism instead of operator new to manufacture instances of the COM class.

Strictly speaking, it's not necessary that the COM class that uses COleObjectFactory as a class factory be dynamically creatable. OnCreateObject() is virtual, so you can override it if you choose to do so. If you choose for some reason to use some other mechanism for creating COM objects, you can simply override COleObjectFactory::OnCreateObject(), using operator new or some other device for allocating the COM object. Though there's usually no good reason to do so, you're free to override OnCreateObject().

COleObjectFactory and IClassFactory::LockServer()

In addition to CreateInstance(), IClassFactory's other member function is LockServer(). LockServer() controls a server's reference count. An MFC module represented by CWinApp contains a reference counter called m_nObjectCount in its AFX_WIN_STATE structure. There's a pair of global MFC functions that increment and decrement the reference count: AfxOleLockApp() and AfxOleUnlockApp(). IClassFactory::LockServer() simply uses AfxOleLockApp() and AfxOleUnlockApp() to manage the application's reference count.

Every time an application calls AfxOleUnlockApp(), AfxOleUnlockApp() decrements the application's object count. If the object count drops to zero, AfxOleUnlockApp() calls AfxOleOnReleaseAllObjects(). AfxOleOnReleaseAllObjects() controls the lifetime of the server. The function first checks to make sure that the user hasn't taken control of the application (for example, if the application is MDI and the user opens a new file). As long as the user hasn't taken control of the application, AfxOleOnReleaseAllObjects() ends the application. If the application has a main window, AfxOleOnReleaseAllObjects() destroys the window. Otherwise, the function posts the WM_QUIT message to end the application.

Creating Class Factories within Your App

MFC makes it easy to add class factories to your application. There are two macros that insert a class factory for a specific COM class in MFC. These are the DECLARE_OLECREATE and IMPLEMENT_OLECREATE macro pair. Like the close-cousin macros for dynamic run-time information, dynamic creation, and serialization, the DECLARE... macro goes in your header file, and the IMPLEMENT... macro goes in your source file. To create a class factory for your COM classes, use DECLARE_OLECREATE in your header file and IMPLEMENT_OLECREATE in your CPP file.

Here's DECLARE_OLECREATE():

```
#define DECLARE_OLECREATE(class_name) \
protected: \
    static AFX_DATA COleObjectFactory factory; \
    static AFX_DATA const GUID AFX_CDECL guid; \
```

DECLARE_OLECREATE() for a specific class declares a static instance of COleObjectFactory and a static GUID associated with the class.

Here's IMPLEMENT_OLECREATE():

```
#define IMPLEMENT_OLECREATE(class_name, external_name, l, w1, w2, b1, \
    b2, b3, b4, b5, b6, b7, b8) \
AFX_DATADEF COleObjectFactory class_name::factory(class_name::guid, \
    RUNTIME_CLASS(class_name), FALSE, _T(external_name)); \
const AFX_DATADEF GUID AFX_CDECL class_name::guid = \
    { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } }; \
```

IMPLEMENT_OLECREATE() constructs the class factory and fills the GUID member that was declared by DECLARE_OLECREATE().

```
protected: \
static COleObjectFactory factory; static const GUID __cdecl guid;
```

Here are the class factory macros expanded within the context of CoMath: **IMPLEMENT_OLECREATE**.

```
ColeObjectFactory CoMath::factory(CoMath::guid,
                                  (&CoMath::classCoMath), 0,
                                  "CoMath"); /
const GUID __cdecl CoMath::guid = { 0x06812840, 0x46e9, 0x11ce,
{ 0xae, 0xff, 0xe7, 0x98, 0xa8, 0x72, 0x14, 0x16 } };;
```

COleObjectFactory and Interface Maps

Before leaving the topic of class factories, let's take a closer look at it. A class factory is really just another COM class: we can see that Microsoft does indeed use interface maps to implement their own COM classes. When you look at the class definition of COleObjectFactory, you see it uses the **DECLARE_INTERFACE_MAP()** and the **BEGIN_INTERFACE_PART()**/**END_INTERFACE_PART()** macros. *Ahh*—it must be implementing a COM interface. Closer examination reveals that COleObjectFactory implements **IClassFactory**. The definition uses **IClassFactory** within the **BEGIN_INTERFACE_PART()**, and **IClassFactory**'s functions are prototyped.

Listing 11-23 shows COleObjectFactory after the preprocessor is done with it. Notice how the interface map macros declare a nested class called XClassFactory replete with the **IUnknown** and the **IClassFactory** functions. This shows MFC's interface maps in action!

Listing 11-23. COleObjectFactory after being preprocessed

```
class ColeObjectFactory : public CCmdTarget
{
public:
    static CRuntimeClass classColeObjectFactory;
    virtual CRuntimeClass* GetRuntimeClass() const;

public:
    ColeObjectFactory(const CLSID & clsid,
                      CRuntimeClass* pRuntimeClass,
                      BOOL bMultiInstance,
                      LPCTSTR lpszProgID);

    BOOL IsRegistered() const;
    const CLSID & GetClassID() const;

    BOOL Register();
    void Revoke();
    void UpdateRegistry(LPCTSTR lpszProgID = 0);
```

```
BOOL IsLicenseValid();

static BOOL __stdcall RegisterAll();
static void __stdcall RevokeAll();
static BOOL __stdcall UpdateRegistryAll(BOOL bRegister = 1);

protected:
    virtual CCmdTarget* OnCreateObject();
    virtual BOOL UpdateRegistry(BOOL bRegister);
    virtual BOOL VerifyUserLicense();
    virtual BOOL GetLicenseKey(DWORD dwReserved, BSTR* pbstrKey);
    virtual BOOL VerifyLicenseKey(BSTR bstrKey);

public:
    virtual ~COleObjectFactory();

public:
    COleObjectFactory* m_pNextFactory;

protected:
    DWORD m_dwRegister;
    CLSID m_clsid;
    CRuntimeClass* m_pRuntimeClass;
    BOOL m_bMultiInstance;
    LPCTSTR m_lpszProgID;
    BYTE m_bLicenseChecked;
    BYTE m_bLicenseValid;
    BYTE m_bRegistered;
    BYTE m_bReserved;

public:
    class XClassFactory : public IClassFactory2 {
public:
    virtual ULONG __export __stdcall AddRef();
    virtual ULONG __export __stdcall Release();
    virtual HRESULT __export __stdcall QueryInterface(const IID & iid,
                                                    LPVOID* ppvObj);

    virtual HRESULT __export __stdcall CreateInstance(LPUNKNOWN,
                                                    const IID &,
                                                    LPVOID* );
    virtual HRESULT __export __stdcall LockServer(BOOL);
    virtual HRESULT __export __stdcall GetLicInfo(LPLICINFO);
    virtual HRESULT __export __stdcall RequestLicKey(DWORD, BSTR* );
```

```

virtual HRESULT __export __stdcall CreateInstanceLic(LPUNKNOWN,
                                                    LPUNKNOWN,
                                                    const IID &,
                                                    BSTR,
                                                    LPVOID*);

} m_xClassFactory;
friend class XClassFactory;

private:
    static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[];
protected:
    static const AFX_INTERFACEMAP interfaceMap;
    virtual const AFX_INTERFACEMAP* GetInterfaceMap() const;

friend SCODE __stdcall AfxDllGetClassObject(const CLSID &,
                                             const IID &,
                                             LPVOID*);

friend SCODE __export __stdcall DllGetClassObject(const CLSID &,
                                                 const IID &,
                                                 LPVOID*);
};

}

```

At this point, there's only one piece missing from the COM picture: how MFC implements DllGetClassObject() and DllCanUnloadNow().

Exporting the Class Factory from a DLL

Recall that DllGetClassObject() is called whenever the client makes a call to CoCreateInstance() or CoGetClassObject(). DllGetClassObject() is responsible for exposing CoMath's class factory. Here's DllGetClassObject's prototype:

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv);
```

The first parameter specifies the GUID of the COM class the client is requesting. The second parameter represents the GUID of the interface the client is requesting. Finally, the third parameter is a place in which to place an interface pointer (usually IClassFactory, because this function is commonly used to get a class factory).

Fortunately, MFC provides support for this through the framework. Notice that COleObjectFactory has a friend function: AfxDllGetClassObject(). MFC implements the well-known function for exposing class factories, DllGetClassObject(). MFC's version of the function is named AfxDllGetClassObject(). COleObjectFactory declares AfxDllGetClassObject() as a friend function because AfxDllGetClassObject() is closely tied to the way the framework maintains class factories.

MFC maintains a list of class factories within the CWinApp-derived class. Every time an instance of COleObjectFactory is constructed, MFC adds it to a list of classes maintained by the application. AfxDIIGetClassObject() walks the module's list of class factories looking for the class factory for the COM class specified by the rclsid parameter. If AfxDIIGetClassObject() finds the class factory, it performs a QueryInterface() on the class factory, asking for the interface specified by the riid parameter. If QueryInterface() succeeds, AfxDIIGetClassObject() places that interface pointer within the ppv parameter. If AfxDIIGetClassObject() cannot find the specified class factory, it returns the CLASS_E_CLASSNOTAVAILABLE error code.

DllCanUnloadNow() is called whenever a client application calls CoFreeUnusedLibraries(). The function returns a BOOL value indicating whether or not the library can be freed. Normally, this function checks the number of reference counts on the entire server. DllCanUnloadNow() returns TRUE if the server can be freed and FALSE otherwise. To get this function to work with the framework, simply call AfxDIICanUnloadNow(). Remember, you put a lock on the server using AfxOleLockApp() and removed the lock using AfxOleUnlockApp(). The framework knows about the locks. AfxDIICanUnloadNow() checks with the framework, returning TRUE if the lock count has dropped to zero and FALSE otherwise.

Here's how AfxDIICanUnloadNow() works. AfxDIICanUnloadNow() uses another global function, AfxOleCanExitApp(), to check the lock count. AfxOleCanExitApp() looks at the value of m_nObjectCount, which is maintained as part of the module's state information. If there are outstanding lock counts, then AfxDIICanUnload() returns S_FALSE immediately. Otherwise, AfxDIICanUnloadNow() walks the list of class factories looking for class factory objects with a reference count larger than one. If AfxDIICanUnloadNow() finds a class factory still in use, the function returns S_FALSE. If AfxDIICanUnloadNow() finds that no class factories are in use, the function returns S_OK.

Conclusion

So there it is—MFC's support for COM. Remember, all OLE features happen through COM. To support OLE features, a framework has to implement COM-compliant classes. There are several ways to implement COM-compliant classes, the most common of which are (1) multiple inheritance using one or more interfaces and (2) class composition using nested classes. MFC happens to use the class composition approach. MFC implements IUnknown through CCmdTarget, and any class derived from CCmdTarget is capable of supporting COM interfaces. In addition, MFC supplies two macros to help construct the nested classes: BEGIN_INTERFACE_PART and END_INTERFACE_PART.

To support COM aggregation, CCmdTarget has two IUnknown implementations: one to handle delegating to the controlling IUnknown and one to handle the class's own lifetime control and interface negotiation. The delegating IUnknown functions are CCmdTarget::ExternalAddRef(), CCmdTarget::ExternalRelease(), and CCmdTarget::ExternalQueryInterface(). The nondelegating IUnknown functions are CCmdTarget::InternalAddRef(), CCmdTarget::InternalRelease(), and CCmdTarget::InternalQueryInterface().

CCmdTarget handles the actual QueryInterface() lookup in InternalQueryInterface(). To facilitate the interface lookup, MFC implements a lookup table called an interface map. Each interface implemented by a CCmdTarget-derived class has a corresponding nested class that performs the real work of the interface. The entries into MFC's interface map relate an interface ID (one of those 128-bit-long GUIDs) to the address of the nested class implementing that interface. CCmdTarget accomplishes QueryInterface() by walking the interface map until it either finds a matching interface ID or runs out of interface map entries.

Finally, MFC provides support for class factories through the COleObjectFactory class. COleObjectFactory implements IClassFactory using the standard MFC nested class and interface map approach. However, MFC adds some functionality to COleObjectFactory by supplying run-time class factory registration and registry-updating functions.

If you peruse the MFC source code, you'll see that this is the mechanism used for all MFC/OLE classes. Now you know how to write an in-proc server using MFC and interface maps. Unfortunately, the information about interface maps given in Tech Note 38 (which comes with Visual C++) is a little sketchy. And nowhere are AfxGetClassObject() and AfxDllCanUnloadNow() documented. The information in this chapter is useful if you ever find yourself needing to write a class that implements interfaces not yet supported by MFC. This probably won't happen very often, but it's good to know what to do if you need to.

In addition, the Visual C++ debugger sometimes lands in the middle of the MFC source code (perhaps an assertion failed or something). It's good to know about how this interface stuff works in case the debugger lands you in the middle of MFC code that implements an OLE interface. Those nested classes can be pretty scary looking if you don't know what is going on.

Now that you understand how MFC implements COM, other OLE features like Automation, OLE documents, and OLE controls will make a great deal more sense. The remaining chapters take on these subjects in turn.

Uniform Data Transfer and MFC

This chapter examines OLE's data transfer mechanism, Uniform Data Transfer (UDT). First, we'll look at some of the difficulties involved in doing data transfer with the mechanisms provided under standard Windows. Then we'll examine the IDataObject interface, and look at how to implement UDT Clipboard transfers using MFC, as well as how MFC performs this magic. Then we'll check out OLE drag-and-drop and see how MFC does drag-and-drop.

Some History

Windows is a rich operating system with lots of goodies available for the end user, including a graphical user interface replete with resizable frames, icons, dialog boxes, and everything else a reasonable GUI should have. In addition to all the visible user-interface characteristics, one of Windows' most useful features is its inter-process communication (IPC) facilities. In the old, DOS-based world, it was fairly inconvenient to move data from one program to another. Users often had to resort to some arcane procedure like copying data to a temporary file by hand (sometimes even having to change the format) and reading it in using another format. Fortunately, this isn't necessary under Windows.

Standard Windows provides two fundamental IPC mechanisms: the Clipboard and dynamic data exchange. Though these mechanisms are infinitely better than anything provided by straight DOS applications, they are far from perfect. In addition to these standard methods of transferring data, you can always share data via a DLL. However, all these methods have some inherent problems. Because of the difficulties with these mechanisms (which will be highlighted shortly), Microsoft provides a standard data transfer mechanism as a feature of OLE: Uniform Data Transfer.

To provide background and context, let's start by taking a quick look at how Windows data transfer was done before OLE.

The Old Way

The ability to transfer data within an application and between two applications is a very important aspect of any modern-day operating system. No single program lays claim to the best features across the board. For example, many word processors are great at editing text. However, they often leave much to be desired as far as editing graphics is concerned. In cases like this, you want to be able to lift a picture out of your drawing program and paste it into your document.

If you're using Windows, there are two standard ways to transfer data within and between applications: by using the Windows Clipboard and by using dynamic data exchange.

The Clipboard is one of the handiest features of Windows, and one of the easiest to use. Using the Windows Clipboard, it is a simple matter to transfer many different types of data within a single application and even between different applications. Almost every single Windows application available today supports at least copy and paste operations on the main menu (usually under the Edit menu). When you select Copy from the menu, the application usually copies the designated data or text on the screen to the Windows Clipboard. Then you can go to almost any other application and select the Paste option. Paste normally takes the information on the Clipboard and inserts it into the current application.

Programming the Windows Clipboard

Adding support for the Windows Clipboard is fairly straightforward. Let's quickly cover how to program the Windows Clipboard, because it's important in understanding the OLE Clipboard. To copy data to the Clipboard, just follow these steps:

1. Call `OpenClipboard()`. This tells the Clipboard which window owns the data.
2. Call `EmptyClipboard()` to purge the Clipboard of any old data.
3. Call `SetClipboardData()` to put data on the Clipboard. At this point, you have two choices: you can actually copy data on the Clipboard or you can promise the Clipboard that you have the data and will render it on demand. `SetClipboardData()` takes two parameters: a `CLIPFORMAT` constant (you know, `CF_TEXT`, `CF_BITMAP`, and so on) and a data handle. If you give a non-NULL handle when calling `SetClipboardData()`, the Windows Clipboard retains the data on the Clipboard. If you call `SetClipboardData()` using a NULL pointer, then you are promising the Clipboard that you have the data and will render it on demand.
4. Call `CloseClipboard()` so that other applications can access the Clipboard.

Once data is made available through the Clipboard, some other application may want to use it. To retrieve the Clipboard data, the data consumer must perform these steps:

1. Call OpenClipboard() to access the Clipboard.
2. Call GetClipboardData() to retrieve the data.
3. Call CloseClipboard() to release the Clipboard.

If the Windows Clipboard is holding the data, then it's no problem. The Clipboard passes the data back to the consumer through the GetClipboardData() function. If the producer elected to use delayed rendering (by calling SetClipboardData() using a NULL handle), then Windows sends a WM_RENDERFORMAT message to the data producer. The data producer responds to the WM_RENDERFORMAT message by producing the data in the requested format.

DDE (a.k.a. Dilapidated Data Exchange)

DDE is the second standard way to transfer data in Windows. When using DDE, two applications set up a conversation between themselves, thereby enabling data exchange. Two applications involved in a DDE conversation can exchange raw data, can notify each other about data changes, and can even have one application execute commands in another. Unfortunately, there are several limitations to the standard Windows data transfer mechanisms.

Limitations of the Windows Clipboard and DDE

One fundamental problem with both the Clipboard and DDE is that they rely on global memory to transfer data. This gets to be a problem, especially when you try to transfer items (such as a device-independent bitmap). When you use a global memory handle to transfer data, Windows has to rely on virtual memory management to accomplish large allocations, which may result in Windows having to swap data out to your hard disk. Then it's coffee break time as your hard disk sits there churning away.

Another significant problem with data transfer using the Windows Clipboard is that there is only one way to describe the data on the Clipboard: through a single 16-bit integer. Windows provides several standard Clipboard formats (such as CF_TEXT or CF_BITMAP), and you can register your own formats. So while you can always specify the format of the data, there's no way to specify *how the data should be rendered*.

Unfortunately, the first version of OLE wasn't much of an improvement. Because OLE1's underlying data transfer mechanism used DDE, it had the same problems

as regular DDE. Now OLE includes a data transfer technology. OLE's protocol—Uniform Data Transfer—solves the data transfer problem by providing a single, unified way of transferring data within and between applications (hence the term *Uniform*). It's uniform because it all happens through **IDataObject** (which is just another COM interface).

OLE and Uniform Data Transfer

The original reason for OLE Uniform Data Transfer was to support OLE documents. Apart from being used in OLE documents, Uniform Data Transfer is also useful for implementing copying and pasting to and from the Clipboard and for doing OLE drag-and-drop. Let's examine **IDataObject**, the main interface behind UDT.

Important Structures

Using OLE to perform data transfer through the Clipboard and for drag-and-drop is fairly straightforward. However, before examining data transfer via the Clipboard, let's take a quick look at two very important UDT-related structures: **FORMATETC** and **STGMEDIUM**. The **FORMATETC** structure describes the actual data involved in the transfer. **STGMEDIUM** describes the medium used in the data transfer. UDT allows data transfer via a large number of mediums (not just global memory).

The **FORMATETC** Structure

The **FORMATETC** structure is a much richer way of describing data than the standard Windows Clipboard format. In addition to describing the format of the data, the structure can also supply information about how the data should be rendered. This includes such information as the intended target device and the aspect (for example, whether to render the data as a bitmap or a thumbnail sketch). The **FORMATETC** structure looks like this:

```
typedef struct tagFORMATETC {
    CLIPFORMAT      cfFormat;
    DVTARGETDEVICE * ptd;
    DWORD           dwAspect;
    LONG            lindex;
    DWORD           tymed;
} FORMATETC;
```

The first field is a Clipboard format (CLIPFORMAT). This can be any of the standard Clipboard formats, such as CF_TEXT and CF_BITMAP. It can also indicate a private application format (as registered with RegisterClipboardFormat()), or it can represent an OLE Clipboard format such as CF_EMBEDSOURCE or CF_LINKSOURCE. Notice that FORMATETC includes a space for all the old Clipboard formats (in the cfFormat field). However, FORMATETC represents more than just the Clipboard format of the data involved in the transfer.

The second field is a pointer to a structure containing information about the device for which the data is rendered. This usually represents a printer, but it can actually describe any device.

The third field, dwAspect, describes the way in which to show the data. Values for dwAspect come from the DVASPECT enumeration:

```
typedef enum tagDVASPECT
{
    DVASPECT_CONTENT      = 1,
    DVASPECT_THUMBNAIL    = 2,
    DVASPECT_ICON          = 4,
    DVASPECT_DOCPRINT      = 8,
} DVASPECT;
```

Even though DVASPECT values are individual flag bits, dwAspect can represent only one value. Table 12-1 lists the meanings of each member of the enumeration.

Table 12-1. The Aspect enumeration describing how much detail to include in the rendering

Value	Meaning
DVASPECT_CONTENT	Indicates a representation of the object when the object is displayed as embedded inside a container. This is the most common value for compound-document objects. You can use DVASPECT_CONTENT to get a presentation of the embedded object for rendering either on the screen or on a printer.
DVASPECT_DOCPRINT	Indicates that the object should be shown as though it were printed.
DVASPECT_THUMBNAIL	Indicates a thumbnail representation of the object. The representation is usually a 120-by-120 pixel, 16-color (recommended), device-independent bitmap wrapped in a metafile. This is useful when displaying the object in a browsing tool.
DVASPECT_ICON	Indicates an iconic representation of the object.
DVASPECT_DOCPRINT	Indicates the object as though it were printed using the Print command from the File menu. The described data represents a sequence of pages.

The fifth field, lindex, has a meaning dependent on the value of dwAspect. If dwAspect is either DVASPECT_ICON or DVASPECT_THUMBNAIL, the lindex is ignored. If dwAspect is DVASPECT_CONTENT or DVASPECT_DOCPRINT, then lindex represents the part of the aspect that is of interest. For example, it might represent the number of pages represented by the data.

The final field, tymed, refers to the type of medium holding the data. The OLE header files define an enumeration called TYMED that represents various types of media used during a data transfer. The TYMED enumeration contains the following values: TYMED_HGLOBAL, TYMED_FILE, TYMED_ISTREAM, TYMED_ISTORAGE, TYMED_GDI, TYMED_MFPICT, TYMED_ENHMF, and TYMED_NULL.

During data transfers, *data producers* are responsible for allocating data, while *data consumers* are responsible for releasing data. Each type of medium requires its own particular release mechanism that must be called when done.

A value of TYMED_HGLOBAL indicates that data is represented by a global memory handle. The global handle must be allocated using the GMEM_SHARE flag. The release mechanism is GlobalFree().

If the second value, TYMED_FILE, is used, then the data transfer is via the contents of a disk file. The release mechanism is to call either OpenFile() using the OF_DELETE flag or DeleteFile(), if you're supporting long file names.

TYMED_ISTREAM denotes that the data transfer method uses an instance of the IStream interface. The release mechanism is the interface's Release() function. For example, if the interface pointer is pStream, the release mechanism is pStream->Release().

The fourth value in the enumeration, TYMED_ISTORAGE, means that the data transfer medium is implemented via the IStorage interface. The release mechanism is the interface's Release() function. For instance, if the interface pointer is pStorage, the release mechanism is pStorage->Release().

The fifth value is TYMED_GDI, which indicates that the data is being passed through a GDI handle, such as a bitmap or a palette. The release mechanism is the API function DeleteObject().

TYMED_MFPICT value means that the data is being passed using the CF_METAFILEPICT format, which contains a nested handle. The release mechanism for TYMED_MFPICT involves (1) locking down the metafile handle (using GlobalLock()), (2) calling DeleteMetaFile(), (3) unlocking the handle, and finally (4) freeing the handle using GlobalFree().

The STGMEDIUM structure, described next, also uses the TYMED enumeration.

The STGMEDIUM Structure

In addition to the FORMATETC structure for describing data, OLE defines another structure, called STGMEDIUM, for holding the data. STGMEDIUM is a tagged union that describes the data. The structure looks like this:

```
typedef struct tagSTGMEDIUM {
    DWORD          tymed;
    union {
        HANDLE      hGlobal;
        LPSTR      lpszFileName;
        IStream*    pstm;
        IStorage*   pstg;
    };
} STGMEDIUM;

IUnknown * punkForRelease;
```

The first field is a DWORD that describes the type of medium being used in the transfer. This field may contain one of the values from the TYMED enumeration described earlier.

The second field is a union that contains the representation of the storage medium (that is, a handle or an interface pointer).

Notice that the STGMEDIUM also contains a pointer to an IUnknown interface called punkForRelease. If the pointer is valid, then the data consumer should call pUnkForRelease->Release(). If the pointer is NULL, then the data consumer is responsible for freeing the data via the appropriate method. For example, if the medium for data is a global memory handle, the data consumer should call GlobalFree() when it's done using the data.

Remember that each type of medium requires its own release mechanism. This can get fairly confusing. Fortunately, OLE provides a wrapper function, called `ReleaseStgMedium()`, that releases data in a `STGMEDIUM` structure. `ReleaseStgMedium()` takes a pointer to a `STGMEDIUM` structure and releases the data in whatever manner is appropriate.

The **IDataObject** Interface

Lying at the heart of UDT is a single interface called `IDataObject`. `IDataObject` is just another COM interface that you may choose to implement (or not to implement, for that matter). Here's the definition of `IDataObject`:

```

virtual HRESULT SetData(FORMATETC* pformatetc, STGMEDIUM* pmedium,
                       BOOL fRelease) = 0;
virtual HRESULT EnumFormatEtc(DWORD wDirection, IEnumFORMATETC** 
                           penumformatetc) = 0;
virtual HRESULT DAdvise(FORMATETC* pformatetc, DWORD grfAdvf,
                       IAdviseSink* pAdvSink, DWORD* pdwConnection) = 0;
virtual HRESULT DUnadvise(DWORD dwConnection) = 0;
virtual HRESULT EnumDAdvise(IEnumSTATDATA** ppenumAdvise) = 0;
};

}

```

As with all COM interfaces, this one is derived from **IUnknown**. In addition to the **IUnknown** functions, **IDataObject** has nine functions for performing data transfer operations. Let's look at each of them very briefly.

IDataObject::GetData()

Naturally, an interface supporting data transfer needs to have a function for retrieving the data. **GetData()** retrieves data from the object implementing **IDataObject**. Notice **GetData()**'s parameters: a pointer to a **FORMATETC** structure and a pointer to a **STGMEDIUM** structure. **IDataObject::GetData()** retrieves the data as specified in the **FORMATETC** structure (if it's available, of course) and passes it back in the **STGMEDIUM** structure.

IDataObject::GetDataHere()

IDataObject::GetDataHere() is similar to **IDataObject::GetData()**, with the exception that the caller provides the actual medium instead of the callee.

IDataObject::QueryGetData()

IDataObject::QueryGetData() allows the caller to determine whether data is available in the format and medium as described by **FORMATETC**.

IDataObject::GetCanonicalFormatEtc()

In some cases, a given data object returns exactly the same data for multiple **FORMATETC** structures. **IDataObject::GetCanonicalFormatEtc()** provides a mechanism whereby the object can communicate to the caller which **FORMATETCs** produce the same output data.

IDataObject::SetData()

`IDataObject::SetData()` is the complement to `IDataObject::GetData()`. It's used to poke data into a data object.

IDataObject::EnumFormatEtc()

`IDataObject::EnumFormatEtc()` is used by the caller to enumerate the formats in which data can be stored into or retrieved from this object with `SetData()` and `GetData()`, respectively.

IDataObject::DAdvise()

`IDataObject::DAdvise()` sets up an advisory connection between the data object and an advisory sink, through which the sink can be informed when data provided by the object later changes.

IDataObject::DUnadvise()

`IDataObject::DUnadvise()` disconnects an advisory connection set up previously with `IDataObject::DAdvise()`.

IDataObject::EnumDAdvise()

`IDataObject::EnumDAdvise()` enumerates the advisory connections currently found on this object.

So that's OLE's data transfer interface, `IDataObject`. Let's see how the OLE Clipboard uses it at the raw level.

The OLE Clipboard

OLE extends the standard Windows Clipboard model. In addition to the standard Windows Clipboard API, the standard Clipboard has been supplemented by the OLE data transfer mechanism. The OLE Clipboard is backwardly compatible with the regular Windows Clipboard. In fact, the OLE Clipboard represents a layer between applications and the regular Windows Clipboard. (See Figure 12-1.)

OLE Clipboard transfers are implemented via the `IDataObject` interface. Four API functions support the OLE Clipboard:

- `OleSetClipboard()`—Puts an `IDataObject` pointer on the Clipboard.
- `OleGetClipboard()`—Retrieves an `IDataObject` from the Clipboard.
- `OleFlushClipboard()`—Clears the OLE Clipboard, releasing the `IDataObject` pointer on the Clipboard.
- `OleIsCurrentClipboard()`—Determines whether a specific data object is the one currently on the Clipboard.

Here's how the OLE Clipboard works: An application that has data to put on the Clipboard implements `IDataObject`. This is the data producer. The data producer calls `OleSetClipboard()`, which makes a copy of `IDataObject` pointer and places it on the OLE Clipboard.

Once the `IDataObject` pointer is on the Clipboard, OLE starts using the Windows Clipboard as a normal Windows program would. OLE calls `OpenClipboard()` to claim ownership of the Clipboard. The OLE Clipboard employs delayed rendering of Clipboard formats. Of course, `OpenClipboard()` requires a window handle. The OLE Clipboard actually owns a window that's created during a call to `OleInitialize()`. (If you want to use the OLE Clipboard, you need to call `OleInitialize()`—`CoInitialize()` doesn't create that window.)

Then OLE enumerates the `IDataObject` formats, calling `SetClipboardData()` for each format that provides data in a global handle. Remember, this is OLE talking to the standard Clipboard. The standard Clipboard doesn't support files or Structured Storage, so it can put only global handles on the Clipboard. When OLE calls `SetClipboardData()`, it does so using a NULL pointer. This indicates to the Windows Clipboard that delayed rendering is in effect.

Sometime later, a data consumer decides to access the Clipboard. If the data consumer isn't OLE savvy, it can access the Clipboard in the regular Windows manner by calling `GetClipboardData()`. Because OLE used delayed rendering, it's time for the producer to pay up. The OLE Clipboard accomplishes this by looking at the `IDataObject` pointer on the Clipboard and calling its `GetData()` method to retrieve the data.

If the data consumer is OLE savvy, the consumer can just call `OleGetClipboard()` to retrieve the `IDataObject` pointer from the Clipboard. Then the consumer can call `IDataObject::GetData()` to retrieve the data. The OLE Clipboard actually maintains its own `IDataObject` pointer. If the data producer put an `IDataObject` pointer on the Clipboard, then the Clipboard's `IDataObject` makes direct calls to the data producer's `IDataObject`. If the data producer did not put an `IDataObject` pointer on the Clipboard, then the OLE Clipboard's `IDataObject` accesses the Clipboard in the regular Windows way by sending `WM_RENDERDATA` messages to the window that owns the Clipboard.

Now that you understand how the OLE Clipboard works, let's take a look at how MFC deals with `IDataObject` and OLE Clipboard transfers.

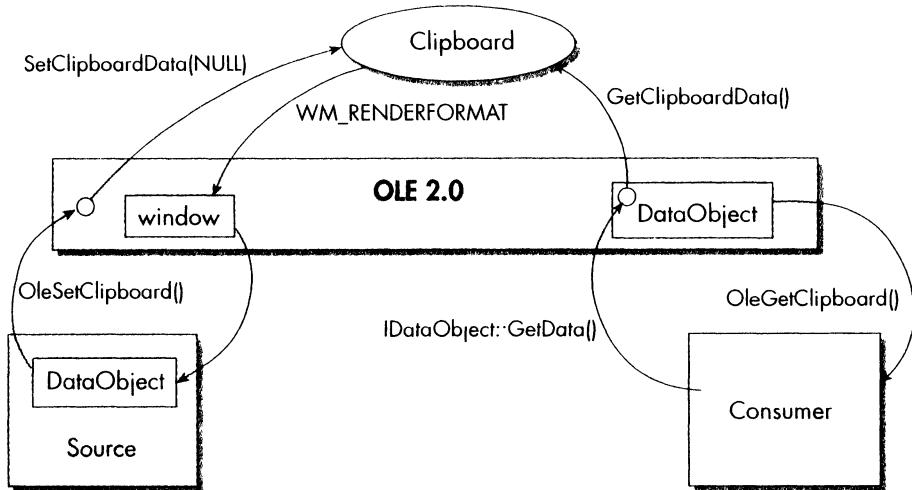


Figure 12-1 The OLE Clipboard

MFC's IDataObject Classes

MFC contains two classes that support the `IDataObject` interface: `COleDataSource` and `COleDataObject`. The main difference between these two interfaces is how they support `IDataObject`. `COleDataSource` is a fully qualified COM object. If you look at the header file and source code listings, you'll see that `COleDataSource` has interface maps, implying that `COleDataSource` is a real COM object. `COleDataObject` is a bit different: it wraps an existing `IDataObject` pointer, giving it a very developer-friendly C++ interface.

As its name implies, `COleDataSource` is normally used on the *producer* side of a data transfer. `COleDataObject` is usually used by the data *consumer*. Before digging into the details, let's see how you might use these objects to do Clipboard data transfers.

Transferring Data via the Clipboard

Using the OLE Clipboard to copy and paste data in an MFC program is quite easy. It involves using the two `IDataObject`-oriented MFC classes: `COleDataSource` and `COleDataObject`. Use `COleDataSource` wherever you originate or produce data. It's the source side of the equation. Use `COleDataObject` on the receiving side whenever you want to consume data in an MFC program. The steps for copying and pasting using the OLE Clipboard are outlined in the following paragraphs.

Copying to the Clipboard

To copy data to the Clipboard, start by getting a pointer to the data you want to put on the Clipboard. For instance, if you want to copy a bit of text to the Clipboard, get a pointer to the string. Then create an instance of the COleDataSource class. You'll use this object to store the data until a data consumer decides to accept it.

Imagine that you want to transfer the data via global memory. In this case, use GlobalAlloc() to allocate a global memory handle. Use GlobalLock() to obtain a live pointer to the memory so you can copy the data into the buffer. Be sure to GlobalUnlock() the handle after copying data into the buffer. Place the data to the data source object by calling the data source's CacheGlobalData(). CacheGlobalData() takes two parameters: the Clipboard format of the data (that is, CF_TEXT, CF_BITMAP, and so on) and the handle to the memory holding the data. Finally, call SetClipboard() to place the data object on the Clipboard. Now another application can use the data on the Clipboard. The following code snippet shows the ease with which you can implement copying data to the Clipboard.

```
{
    LPCSTR source = GetString();

    COleDataSource *pcods;
    pcods = new COleDataSource;
    HGLOBAL h =
        GlobalAlloc(GHND | GMEM_SHARE, strlen(source) + 1);
    strcpy(LPSTR(GlobalLock(h)), source);
    GlobalUnlock(h);
    pcods->CacheGlobalData(CF_TEXT, h);
    pcods->SetClipboard();
}
```

You may notice something funky in this code. The function allocates a new COleObject but never deletes it. What's going on? Buried in the MFC documentation, Microsoft says, "In situations where you hand the data source to OLE, such as calling COleDataSource::SetClipboard(), you do not need to worry about destroying it because it will be destroyed by OLE." Here's what happens. By calling COleDataSource::SetClipboard(), you hand the responsibility of managing the object implementing IDataObject over to OLE (in this case, the object is your COleDataSource object). Anytime an application calls the OLE API function ::OleSetClipboard(), OLE cleans out the Clipboard, releasing the IDataObject pointer on the Clipboard if it's there. Of course, Release() will delete the object (provided there are no outstanding reference pointers). Your COleDataSource has to stick around as long as some data consumer may want to get data out of it.

If you want to use a medium other than global memory, that's equally easy. Again, create an instance of the COleDataSource class. However, instead of using COleDataSource::CacheGlobalData() to place the data in the data source object, use COleDataSource::CacheData(). COleDataSource::CacheData() is much more flexible than COleDataSource::CacheGlobalData(). Instead of taking a single Clipboard format and global memory handle, COleDataSource::CacheData() takes pointers to a FORMATETC structure and to a STGMEDIUM structure. The FORMATETC structure describes the format of the data, and the STGMEDIUM structure specifies the transfer medium for the data. Once the data is copied to the Clipboard, then any other application can query the Clipboard and use the data. The process of retrieving data from the Clipboard is called pasting.

Pasting from the Clipboard

Retrieving data from the Clipboard is as easy as copying data to the Clipboard using COleDataSource. The framework uses COleDataObject to pass data to the application on the consumer side of the transfer.

To paste data from the Clipboard, declare an instance of COleDataObject and retrieve data from the Clipboard by calling COleDataObject::AttachClipboard(). Once the Clipboard data is attached to COleDataObject, find out what format the data is in by using one of two methods: (1) calling COleDataObject::IsDataAvailable() or (2) enumerating the formats.

Discovering Available Data

COleDataObject::IsDataAvailable() takes two arguments: a CLIPFORMAT and a pointer to a FORMATETC structure. If you want to interrogate the data object for one of the standard Clipboard formats (such as CF_TEXT or CF_BITMAP), supply that format in the first parameter and leave the pointer to the FORMATETC structure NULL. If you want to use more detailed information when interrogating the Clipboard, fill in a FORMATETC structure and pass it to IsDataAvailable(). IsDataAvailable() returns a BOOL indicating whether or not the COleDataObject is holding the data in a format you're interested in.

The other way of finding out what kind of data is available is to enumerate the COleDataObject's Clipboard formats. COleDataObject includes two functions for doing this: BeginEnumFormats() and GetNextFormat(). BeginEnumFormats() starts the enumeration process. Once you call BeginEnumFormats(), you can begin retrieving the available formats one at a time using GetNextFormat(). GetNextFormat() fills in a FORMATETC structure describing the format of the data. If GetNextFormat() returns TRUE, then data is available. GetNextFormat() fills the FORMATETC structure with the information about the data format.

Retrieving the Data

Once you determine if data is available and what kind it is, the next step is to retrieve it. COleDataObject provides three functions for retrieving data: GetData(), GetFileData(), and GetGlobalData().

COleDataObject::GetData() offers the most generic way of retrieving data from the data object. GetData() takes three parameters: a CLIPFORMAT, a pointer to a STGMEDIUM structure, and a pointer to a FORMATETC structure. The CLIPFORMAT specifies the format in which the data is returned. GetData returns the data in the STGMEDIUM structure parameter. Finally, the FORMATETC describes data format information above and beyond the format specified by the CLIPFORMAT parameter.

If you know that the storage medium used by the COleDataObject is a file, you can use COleDataObject::GetFileData(). GetFileData() takes only two parameters for describing the format of the data to be retrieved: a CLIPFORMAT identifier and a pointer to a FORMATETC structure. GetFileData() returns a pointer to a CFile object that contains the data held by the COleDataObject. Note that the CFile returned could be a COleStreamFile, CSharedFile, or a normal CFile, depending on the medium that the data is actually stored on. In addition, the caller of GetFileData() is responsible for deleting the CFile* when it is done with it.

Finally, if you know that the storage medium of the data you want is a global memory handle, you can call COleDataObject::GetGlobalData() to retrieve the data. Like GetFileData(), GetGlobalData() takes two parameters describing the format of the data being retrieved: a CLIPFORMAT and a pointer to a FORMATETC structure. GetGlobalData() returns a global handle that you can lock with GlobalLock() to retrieve a live pointer. The following code snippet shows how to use a COleDataObject to pull data off the Clipboard and use it.

```
ColeDataObject codo;

codo.AttachClipboard();

if (codo.IsDataAvailable(CF_TEXT)) {
    HANDLE h = codo.GetGlobalData(CF_TEXT);

    CEdit& rEditCtrl = GetEditCtrl();

    // Use data

    GlobalUnlock(h);
    GlobalFree(h);

    GetDocument()->UpdateAllViews(0);
    GetDocument()->SetModifiedFlag(TRUE);
}

codo.Release();
```

Delayed Rendering

Clipboard data transfer using the preceding methods is an immediate rendering technique. That is, when the data producer puts data on the Clipboard, the data is available immediately when the data consumer asks for data. That's because COleDataSource's caching functions maintain the data for the life of the COleDataSource object.

Delayed rendering is another data transfer technique, a scenario in which the data producer isn't required to render the data until it's demanded by the consumer. Delayed rendering is particularly useful when you want to transfer very large amounts of data. It may be very expensive to keep a huge block of data in memory for long periods of time (as would happen by caching the data in the COleDataSource). A better solution is to promise the data and create the data block only on demand. That way you don't need to expend system resources by maintaining the huge block of data. This technique is enabled through three of COleDataSource's member functions: DelayRenderData(), DelayRenderFileData(), and OnRenderData().

To implement delayed rendering, simply derive a class from COleDataSource. You need to derive your own class because you need to override OnRenderData().

When a data producer calls DelayRenderData(), the data producer promises that it has data available in a particular format by passing a CLIPFORMAT and a FORMATETC structure describing the data format. This function should be used to delay rendering when using storage mediums other than a CFile object.

DelayRenderFileData() works just like DelayRenderData(); however, it's used specifically for delaying rendering of data using a CFile object as its storage medium. It also takes a CLIPFORMAT and a FORMATETC structure as arguments.

So how does the data actually get to the data consumer in a delayed rendering situation? COleDataSource provides three functions for supplying data on demand: OnRenderData(), OnRenderFileData(), and OnRenderGlobalData(). The derived class overrides these functions.

Whenever a data consumer requests data supplied through either DelayRenderData() or DelayRenderFileData(), MFC calls the data producer through one of COleDataSource's three OnRender() functions: OnRenderData(), OnRenderFileData(), or OnRenderGlobalData(). These are virtual functions that you override to supply data to the data consumer.

OnRenderData() is called by the framework to retrieve data in the format specified by the CLIPFORMAT and/or FORMATETC structure represented as OnRenderData()'s parameters. OnRenderData() also takes a pointer to a STGMEDIUM structure, which specifies the storage medium used for the data transfer. To provide the data, examine the STGMEDIUM's tymed field to determine what storage medium the data consumer is requesting. Then copy the data to the requested medium and set the appropriate field in the STGMEDIUM's union to the requested medium.

The framework calls `OnRenderFileData()` whenever a data consumer wants to retrieve data that was placed on the Clipboard by a call to the `COleDataSource`'s `DelayRenderFileData()` member function. As with `OnRenderData()`, `OnRenderFileData()` takes data format information through a single `FORMATETC` parameter. Of course, there's no `STGMEDIUM` structure in this case because this function specifically uses files as the transfer medium. The framework passes a pointer to a `CFile` object, which can be set to hold your data.

The last `OnRender...()` function is `OnRenderGlobalData()`. This function is very similar to `OnRenderFileData()` except that the framework calls this function to render explicitly a global memory handle. So, instead of a pointer to a `CFile` object (as with `OnRenderFileData()`), the framework passes a pointer to an `HGLOBAL`, which can be used to pass a global handle to the data consumer.

Please note that you need to be somewhat careful about the `HGLOBAL*` or `STGMEDIUM*` parameter passed in. In particular, there may already be a non-NULL `HGLOBAL` there. The `OnRender()` functions are called in both the `GetData()` and `GetDataHere()` cases. In the `GetDataHere()`, the caller provides a place to put the data. In the `GetData()` case the `STGMEDIUM/HGLOBAL` are not filled in already.

For example, imagine you promised data consumers that you can provide a piece of data by calling `COleDataSource::DelayRenderData()`. A data consumer may then ask your program to provide data via a global memory handle (by calling `COleDataObject::GetGlobalData()`). When the consumer tries to retrieve the data from the `COleDataObject`, the framework calls your `OnRenderGlobalData()` function. Inside your `OnRenderGlobalData()` function, copy your data to a global memory handle and set the `HGLOBAL` parameter to point to your global memory handle (if the `HGLOBAL` is not already set).

A point of interest: the technique for rendering data (immediately or delayed) is transparent to the data consumer. When a data consumer retrieves data from the Clipboard (using the regular methods for pasting data discussed previously), the consumer has no idea whether the data is being supplied directly or indirectly (that is, whether or not the data is rendered immediately or on a delayed basis).

MFC's `IDataObject` Classes in Detail

In summary, doing Clipboard data transfers in MFC is pretty easy. Either (1) cache the data in an instance of `COleDataSource` or (2) call one of the delayed rendering functions and handle the `OnRender...()` functions. Let's take a look under the hood.

COleDataSource

Listing 12-1 shows the definition of COleDataSource from AFXOLE.H:

Listing 12-1. The COleDataSource class

```
class COleDataSource : public CCmdTarget {
// Constructors
public:
    COleDataSource();

// Operations
    void Empty();

    // CacheData & DelayRenderData operations similar to ::SetClipboardData
    void CacheGlobalData(CLIPFORMAT cfFormat, HGLOBAL hGlobal,
        LPFORMATETC lpFormatEtc = NULL);      // for HGLOBAL based data
    void DelayRenderFileData(CLIPFORMAT cfFormat,
        LPFORMATETC lpFormatEtc = NULL);      // for CFile* based delayed render

    // Clipboard and Drag/Drop access
    DROPEFFECT DoDragDrop(
        DWORD dwEffects = DROPEFFECT_COPY|DROPEFFECT_MOVE|DROPEFFECT_LINK,
        LPCRECT lpRectStartDrag = NULL,
        COleDropSource* pDropSource = NULL);
    void SetClipboard();
    static void PASCAL FlushClipboard();
    static COleDataSource* PASCAL GetClipboardOwner();

    // Advanced: STGMEDIUM based cached data
    void CacheData(CLIPFORMAT cfFormat, LPSTGMEDIUM lpStgMedium,
        LPFORMATETC lpFormatEtc = NULL);      // for LPSTGMEDIUM based data
    // Advanced: STGMEDIUM or HGLOBAL based delayed render
    void DelayRenderData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);

    // Advanced: support for SetData in ColeServerItem
    // (not generally useful for Clipboard or drag/drop operations)
    void DelaySetData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);

// Overridables
    virtual BOOL OnRenderGlobalData(LPFORMATETC lpFormatEtc,
                                    HGLOBAL* phGlobal);
    virtual BOOL OnRenderFileData(LPFORMATETC lpFormatEtc, CFile* pFile);
    virtual BOOL OnRenderData(LPFORMATETC lpFormatEtc,
                            LPSTGMEDIUM lpStgMedium);
    // OnRenderFileData and OnRenderGlobalData are called by
    // the default implementation of OnRenderData.
```

```

virtual BOOL OnSetData(LPFORMATETC lpFormatEtc, LPSTGMEDIUM lpStgMedium,
    BOOL bRelease);
    // used only in COleServerItem implementation

// Implementation
public:
    virtual ~COleDataSource();

protected:
    AFX_DATACACHE_ENTRY* m_pDataCache; // data cache itself
    UINT m_nMaxSize; // current allocated size
    UINT m_nSize; // current size of the cache
    UINT m_nGrowBy; // number of cache elements to grow by for new allocs

    AFX_DATACACHE_ENTRY* Lookup(
        LPFORMATETC lpFormatEtc, DATADIR nDataDir) const;
    AFX_DATACACHE_ENTRY* GetCacheEntry(
        LPFORMATETC lpFormatEtc, DATADIR nDataDir);

// Interface Maps
public:
    BEGIN_INTERFACE_PART(DataObject, IDataObject)
        INIT_INTERFACE_PART(COleDataSource, DataObject)
        STDMETHOD(GetData)(LPFORMATETC, LPSTGMEDIUM);
        STDMETHOD(GetDataHere)(LPFORMATETC, LPSTGMEDIUM);
        STDMETHOD(QueryGetData)(LPFORMATETC);
        STDMETHOD(GetCanonicalFormatEtc)(LPFORMATETC, LPFORMATETC);
        STDMETHOD(SetData)(LPFORMATETC, LPSTGMEDIUM, BOOL);
        STDMETHOD(EnumFormatEtc)(DWORD, LPENUMFORMATETC*);
        STDMETHOD(DAdvise)(LPFORMATETC, DWORD, LPADVISESINK, LPDWORD);
        STDMETHOD(DUnadvise)(DWORD);
        STDMETHOD(EnumDAdvise)(LPENUMSTATDATA*);
    END_INTERFACE_PART(DataObject)

    DECLARE_INTERFACE_MAP()

    friend class COleServerItem;
};

}

```

As you can tell from the header file, COleDataSource is designed specifically to support Clipboard and drag-and-drop transfers (notice the “SetClipboard()” and “DoDragDrop()” functions). We’ll skip the drag-and-drop support for the moment and concentrate on the Clipboard support.

Recall how the OLE Clipboard works. When a data producer decides to put data on the OLE Clipboard, it simply has to implement IDataObject and use OleSetClipboard() to place the IDataObject on the Clipboard.

COleDataSource obviously implements IDataObject: notice the interface map macros. Also notice that COleDataSource has a member function called SetClipboard(). SetClipboard() simply places the COleDataSource's IDataObject interface pointer by calling the global API function ::OleSetClipboard() to put it on the Clipboard. That part is simple. However, there's still the issue of getting the data into the COleDataSource. That's done using COleDataSource's caching functions, CacheData() and CacheGlobalData().

Caching the Data

The idea behind caching the data is that COleDataSource keeps the data around so that it can render the data immediately upon demand. That implies that the COleDataSource utilizes some sort of data structure to manage data that's been cached. In fact, COleDataSource uses a structure called AFX_DATACACHE_ENTRY to store the cached data. Here's the definition of the AFX_DATACACHE_ENTRY:

```
struct AFX_DATACACHE_ENTRY
{
    FORMATETC m_formatEtc;
    STGMEDIUM m_stgMedium;
    DATADIR m_nDataDir;
};
```

There's nothing too complicated here. The structure contains FORMATETC and STGMEDIUM members as well as a DATADIR member. You've already seen the FORMATETC and the STGMEDIUM structures. DATADIR is just an enumeration that establishes the direction of a data transfer. It's defined in OBJIDL.H:

```
enum tagDATADIR {
    DATADIR_GET      = 1,
    DATADIR_SET      = 2
} DATADIR;
```

The DATADIR field is used during delayed rendering, as we'll see shortly.

Take another look at COleDataSource; you'll see that it maintains an array of these AFX_DATACACHE_ENTRY structures. The m_pDataCache member points to the first element in this array. CacheData() and CacheGlobalData() simply add elements to this array.

Inside CacheData() and CacheGlobalData()

COleDataSource::CacheData() and COleDataSource::CacheGlobalData() are similar functions. Both store data in a COleDataSource object so that it may be rendered immediately. CacheData() stores data in a STGMEDIUM structure, while CacheGlobalData()

is a shortcut for caching data in an HGLOBAL. Let's look at the more general function, CacheData(), to see how COleDataSource caches data.

When an application caches data using CacheData(), the application is responsible for packaging the STGMEDIUM structure. In addition, the application has to at least provide a Clipboard format (one of those CF_... constants) indicating the format of the data. If the application wants to define the data using more than the Clipboard format, then the application has to fill in a FORMATETC structure and send that along as part of the CacheData() function.

Notice that COleDataSource::CacheData() includes a FORMATETC structure in the parameter list. This is an optional parameter: the caller can pass just the CLIPFORMAT if it wants to. However, the AFX_DATACACHE_ENTRY structure needs an entire FORMATETC structure—not just a CLIPFORMAT. If the caller doesn't pass in a FORMATETC structure, CacheData() fills the structure using _AfxFillFormatEtc(). _AfxFillFormatEtc() fills the FORMATETC structure using these default values:

```
lpFormatEtc->cfFormat = <the CF_ format provided by the caller>;
lpFormatEtc->ptd = NULL;
lpFormatEtc->dwAspect = DVASPECT_CONTENT;
lpFormatEtc->lindex = -1;
lpFormatEtc->tymed = (TYMED)-1
```

Finally, CacheData() sets the tymed field of the FORMATETC structure to the tymed field of the STGMEDIUM structure passed in by the caller.

Once the framework has a viable FORMATETC, CacheData() makes a place for the data in the cache by calling COleDataSource::GetCacheEntry(). GetCacheEntry() uses the COleDataSource Lookup() to walk the array of AFX_DATACACHE_ENTRY structures in an effort to find an entry that has the same FORMATETC structure as the one the caller is trying to add. If it's already there, Lookup() returns that one. Otherwise, it allocates a new AFX_DATACACHE_ENTRY for the cache array and returns a pointer to the new AFX_DATACACHE_ENTRY. This way, there's no ambiguity regarding the COleDataSource's cache entries. There is at most only one entry for each format described in the FORMATETC structure. Once GetCacheEntry() has a pointer to a valid AFX_DATACACHE_ENTRY, the GetCacheEntry() fills the structure. Of course, if there's already data in the structure, GetCacheEntry() has to release it first (using ReleaseStgMedium()). Once the data's cached in the COleDataSource, data consumers can retrieve it by first getting the IDataObject interface pointer and then calling GetData(). (We'll see exactly how GetData() works shortly.)

Inside Delayed Rendering

When using delayed rendering, data sources are not required to render the data until the very last minute. Delayed rendering happens in much the same way as caching data.

Delayed rendering uses the same AFX_DATACACHE_ENTRY but in a slightly different way.

Like CacheData(), DelayRenderData() takes a CLIPFORMAT and a FORMATETC. However, DelayRenderData() doesn't take a STGMEDIUM. Remember, in this situation, the data producer doesn't have to cough up the data until absolutely necessary, so there's no need for a STGMEDIUM structure.

Just as CacheData() does, DelayRenderData() calls `_AfxFillFormatEtc()`. At that point, DelayRenderData() uses `GetCacheEntry()` to either reuse a cache entry (if one already exists for a specific FORMATETC structure) or to create a new one if necessary. However, instead of adding a STGMEDIUM to the cache, DelayRenderData() zeroes out the STGMEDIUM area of the AFX_DATACACHE_ENTRY. That's how COleDataSource::GetData() knows whether to render the data from the cache directly or to call one of the COleDataSource::OnRenderData() variants.

As you can see, the decision as to when to render the data is made up front. Both CacheData() and DelayRenderData() place an AFX_DATACACHE_ENTRY in the COleDataSource. However, while CacheData() stores the STGMEDIUM structure and the FORMATETC structure, DelayRenderData() stores only the FORMATETC structure.

COleDataSource uses this information when responding to `IDataObject::GetData()`.

Responding to `GetData()`

COleDataSource is a fully qualified COM object that implements `IDataObject`. When clients pull the data down from the Clipboard, COleDataSource's `IDataObject` eventually has to respond to `IDataObject::GetData()`.

COleDataSource's implementation is fairly straightforward. If you look in the MFC source code, you'll see a file called OLEOBJ2.CPP. In it you'll find the code for COleDataSource's `IDataObject::GetData()` function. Of course, COleDataSource implements the interface using the same mechanism as all other MFC COM objects: using interface maps. The actual function is COleDataSource::XDataObject::GetData(). This function walks COleDataSource's data cache looking for an entry that matches the requested format. If the data cache has data in the STGMEDIUM field, then `GetData()` returns the data to the caller using an MFC helper function called `_AfxCopyStgMedium()`.

`_AfxCopyStgMedium()` is a long function (about 160 lines). Whoa! It's very much like the API function `ReleaseStgMedium()`. `_AfxCopyStgMedium()` is a big switch statement that switches on the type of medium being copied. For example, if the type of medium being copied is an HGLOCAL, `_AfxCopyStgMedium()` uses the API function `CopyGlobalMemory()` to make a copy of the data.

If the entry was placed in the cache using delayed rendering, then the data cache entry's STGMEDIUM is empty. In this case, `GetData()` calls COleDataSource::OnRenderData(), at which point the data producer is required to pay up.

Other IDataObject Interface Functions

GetDataHere()

IDataObject::GetDataHere() is just like IDataObject::GetData() except the caller supplies the medium (instead of the object). COleDataSource's implementation uses the Lookup() function to find the appropriate entry in the cache. If Lookup() produces an entry, GetDataHere() uses _AfxCopyStgMedium() to copy the data from the cache entry to the STGMEDIUM provided by the caller.

QueryGetData()

QueryGetData() is simple: it just uses COleDataSource::Lookup() to see if a requested format is in the cache.

GetCanonicalFormatEtc()

There is no canonical FORMATETC for MFC's COleDataSource. Because MFC supports the target device (ptd) for server metafile format, all members of the FORMATETC are significant.

SetData()

SetData() looks in the cache for an entry matching the FORMATETC specified by the caller. If it can find an entry, SetData() calls the virtual function OnSetData(). MFC's default implementation of OnSetData() does nothing. To enable COleDataSource's SetData() function, you need to override OnSetData().

EnumFormatEtc()

Data consumers that want to learn all the formats supplied by a particular data object call EnumFormatEtc(). To accomplish this, COleDataSource supplies an enumerator for the entries in its cache.

COM defines several enumerator interfaces, including IEnumVARIANT, IEnumUnknown, and IEnumFORMATETC. The semantics of enumerating a collection is almost always the same, regardless of the elements involved. Therefore, each of the COM enumerators follows the same form. That is, each enumerator has the same four functions: Next(), Skip(), Reset(), and Clone(). MFC has a helper class defined within OLEIMPL2.H that defines a generic enumerator class called CEnumArray. CEnumArray is another COM object that implements an interface called IEnumVOID. IEnumVOID isn't a standard COM interface: it's defined by MFC in OLEIMPL.H. IEnumVOID enumerates a collection of void pointers. CEnumArray maintains a collection of some arbitrary type and provides all the functionality necessary to enumerate the collection.

Inside OLEDOBJ2.CPP, MFC defines a class called CEnumFormatEtc, which is derived from CEnumArray. CEnumFormatEtc implements an enumerator for

COleDataSource's data cache. COleDataSource implements `EnumFormatEtc()` using `CEnumFormatEtc`. `EnumFormatEtc()` starts declaring an instance of `CEnumFormatEtc`, walking the data cache (the array of `AFX_DATACACHE_ENTRY` structures), and adding each `FORMATETC` to the list maintained by `CEnumFormatEtc`. Then `COleDataSource::EnumFormatEtc()` hands the enumerator interface pointer back to the client.

DAdvise(), DUnadvise(), and EnumDAdvise()

There's not much to say here. COleDataSource is meant to manage Clipboard and drag-and-drop transfers. Each of the advise-loop functions returns `OLE_E ADVISEDNOTSUPPORTED`.

COleDataObject

COleDataObject is usually used on the receiving end of a data transfer. While the COleDataSource class actually implements `IDataObject`, COleDataObject is merely a wrapper around an existing `IDataObject` pointer. Listing 12-2 shows the definition of COleDataObject from the file AFXOLE.H.

Listing 12-2. The COleDataObject class

```
class COleDataObject {
// Constructors
public:
    COleDataObject();

// Operations
    void Attach(LPDATAOBJECT lpDataObject, BOOL bAutoRelease = TRUE);
    LPDATAOBJECT Detach(); // detach and get ownership of m_lpDataObject
    void Release(); // detach and Release ownership of m_lpDataObject
    BOOL AttachClipboard(); // attach to current Clipboard object

// Attributes
    void BeginEnumFormats();
    BOOL GetNextFormat(LPFORMATETC lpFormatEtc);
    CFile* GetFileData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);
    HGLOBAL GetGlobalData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc =
        NULL);
    BOOL GetData(CLIPFORMAT cfFormat, LPSTGMEDIUM lpStgMedium,
        LPFORMATETC lpFormatEtc = NULL);
    BOOL IsDataAvailable(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);

// Implementation
public:
    LPDATAOBJECT m_lpDataObject;
```

```

LPENUMFORMATETC m_lpEnumerator;
~COleDataObject();

// advanced use and implementation
LPDATAOBJECT GetIDataObject(BOOL bAddRef);
void EnsureClipboardObject();
BOOL m_bClipboard;      // TRUE if represents the Win32 Clipboard

protected:
    BOOL m_bAutoRelease;   // TRUE if destructor should call Release

private:
    // Disable the copy constructor and assignment by default so you will get
    // compiler errors instead of unexpected behaviour if you pass objects
    // by value or assign objects.
    COleDataObject(const COleDataObject&); // no implementation
    void operator=(const COleDataObject&); // no implementation
};

}

```

Here's a rundown of the most important COleDataObject functions: Attach(), AttachClipboard(), BeginEnumFormats/GetNextFormat(), and the GetData...() functions.

Attach() and Detach()

Recall that COleDataObject is merely a wrapper for an existing IDataObject pointer. Notice that COleDataObject maintains an IDataObject member, called m_lpDataObject. Like other MFC classes that wrap native things such as handles, COleDataObject has a function for attaching an existing interface pointer to the class. That function is called Attach(). Attach() takes two parameters: a pointer to an IDataObject interface and a flag indicating whether or not the interface pointer should be released when the COleDataObject object is destroyed. COleDataObject::Attach() releases m_lpDataObject (if it is valid and if m_bAutoRelease is TRUE), then sets the m_lpDataObject and the m_bAutoRelease members to the new values provided by parameters.

There's also a complementary function called Detach() that sets m_lpDataObject to NULL. COleDataObject::Release() releases the IDataObject pointer—if m_bAutoRelease is set to TRUE.

AttachClipboard() and EnsureClipboardData()

The documentation provided by Microsoft indicates that if you want to retrieve data from the Clipboard, you should call COleDataObject::AttachClipboard(). Yes, AttachClipboard() is the member you should call. However, AttachClipboard() simply sets COleDataObject's m_bClipboard flag to TRUE. The real action occurs in COleDataObject::EnsureClipboardObject().

`EnsureClipboardObject()` is a wrapper for the API function `OleGetClipboard()`. If the `m_bClipboard` flag (the one set by `AttachClipboard()`) is TRUE, then `OleGetClipboard()` pulls down a copy of the OLE Clipboard's `IDataObject` pointer and wraps the pointer using friendly MFC class member functions provided by `COleDataObject`.

MFC accesses the Clipboard this way as an optimization. You'll see why when examining `COleDataObject::IsDataAvailable()`.

Determining Available Data

`COleDataObject` has a function called `IsDataAvailable()` that takes a `CLIPFORMAT` and a `FORMATETC` as its parameters. `IsDataAvailable()` checks the `m_bClipboard` flag. If the flag is FALSE, then `IsDataAvailable()` calls the `IDataObject` interface's `QueryGetData()` to find out if the data is available in the given format.

If `m_bClipboard` is TRUE, then there's probably data on the Clipboard (that is, the user has called `AttachClipboard()`). `IsDataAvailable()` simply defers to the standard Windows API functions `::IsClipboardFormatAvailable()`, which is often more efficient and reliable than calling `IDataObject::QueryGetData()`.

Notice that `IsDataAvailable()` takes a `CLIPFORMAT` and a `FORMATETC` structure. However, if the consumer has called `AttachClipboard()`, then `IsDataAvailable()` calls the Win32 API that accepts only a `CLIPFORMAT` as a parameter. What if the consumer is requesting data using a `FORMATETC` structure? The Win32 function doesn't know how to deal with a `FORMATETC` structure either. The OLE API ignores everything but the clipformat! Maybe when the Windows API is implemented on top of OLE instead of the other way around, this will change.

Enumerating Formats

Recall that `IDataObject` has a function called `EnumFormatEtc()`. `EnumFormatEtc()` returns a pointer to an interface that enumerates the Clipboard formats. `COleDataObject::BeginEnumFormats()` starts the enumeration by retrieving the enumeration interface pointer from the underlying `IDataObject` interface. To retrieve the actual formats, clients call `COleDataObject::GetNextFormat()`. `GetNextFormat()` uses the enumerator passed back during the call to `COleDataObject::BeginEnumFormats()`, calling the enumerator's `Next()` function.

Retrieving the Data

`GetData()` calls `EnsureClipboardData()` to get the `IDataObject` off the Clipboard, then defers the `IDataObject` pointer's `GetData()` to retrieve the data.

`COleDataObject`'s other `GetData...()` variants are `GetGlobalData` and `GetFileData()`. They work the same way as `GetData()`, except `GetGlobalData()` retrieves a global memory handle and `GetFileData()` retrieves a file handle.

As you can see, MFC's IDataObject classes (COleDataSource and COleDataObject) wrap Clipboard data transfers very well. They also do a great job implementing OLE drag-and-drop. In fact, doing drag-and-drop is very similar and only takes a little bit more work to implement.

OLE Drag-and-Drop

Like OLE Clipboard transfers, OLE drag-and-drop involves getting an IDataObject from a data producer to the consumer. Clearly, the data producer has to implement IDataObject, and the data consumer has to know how to deal with an IDataObject pointer.

In addition to IDataObject, OLE drag-and-drop involves two other interfaces: IDropSource and IDropTarget. Here's the IDropSource interface.

IDropSource

IDropSource handles the user interface for drag-and-drop operations. It's the data producer's job to implement this interface. Here's the actual interface definition:

```
interface IDropSource : public IUnknown {
public:
    virtual HRESULT QueryContinueDrag(BOOL fEscapePressed,
                                      DWORD grfKeyState) = 0;
    virtual HRESULT GiveFeedback(DWORD dwEffect) = 0;
};
```

- QueryContinueDrag() determines whether to continue or to cancel a drag-and-drop operation.
- GiveFeedback() changes the mouse cursor according to the drop effect flags (DROPEFFECT_NONE, DROPEFFECT_MOVE, DROPEFFECT_COPY, and others).

IDropTarget

As its name indicates, IDropTarget handles the target end of a drag-and-drop operation. COM objects that wish to be drop targets implement IDropTarget. At the raw COM level, an OLE API function called RegisterDragDrop() associates a window handle to the COM object implementing IDropTarget. Once a drop target has been registered, OLE can notify the drop target whenever mouse dragging action is occurring

over the window specified in RegisterDragDrop through one of IDropTarget's four member functions, as shown in Listing 12-3.

Listing 12-3. The IDropTarget interface

```
interface IDropTarget : public IUnknown {
public:
    virtual HRESULT DragEnter(IDataObject *pDataObj,
                               DWORD grfKeyState,
                               POINTL pt,
                               DWORD *pdwEffect) = 0;

    virtual HRESULT DragOver(DWORD grfKeyState,
                           POINTL pt,
                           DWORD *pdwEffect) = 0;

    virtual HRESULT DragLeave( void);

    virtual HRESULT Drop(IDataObject *pDataObj,
                        DWORD grfKeyState,
                        POINTL pt,
                        DWORD *pdwEffect) = 0;
};

};
```

- OLE calls IDropTarget::DragEnter() whenever a mouse cursor involved in a drag-and-drop operation enters the window.
- OLE calls IDropTarget::DragOver() multiple times as the mouse cursor representing a drag-and-drop operation moves around over the window.
- OLE calls IDropTarget::DragLeave() whenever the drag-and-drop mouse cursor leaves the window.
- OLE calls IDropTarget::Drop() when the mouse button is released over a drop target window.

Finally, OLE has an API function called DoDragDrop(), which commences the drag-and-drop operation. Once called, DoDragDrop() enters a modal loop, monitoring changes in the mouse and keyboard (especially the ESCAPE key, which cancels it) until the user finishes the drag-and-drop operation by either releasing the mouse button or by pressing the Escape key.

MFC provides a very convenient way to do drag-and-drop transfers: you'll use COleDataSource, COleDataObject, and two other classes, called COleDropTarget and COleDropSource.

Implementing Drag-and-Drop Data Transfer Using MFC

Drag-and-drop is a shortcut for doing Clipboard transfers. Drag-and-drop data transfer is interactive, so its requirements are different from those of Clipboard data transfers. The basic idea behind drag-and-drop data transfers is that you can select the data you want by pressing the left mouse button. Then, as you continue to hold down the left button, you move the mouse cursor (dragging the data) into another window (the one to contain the new data). Once the mouse cursor is over the second window, you drop the data by releasing the left mouse button.

You can further control the kind of transfer by using one or more of the control keys:

<i>Key Pressed</i>	<i>Type of Transfer</i>
No control keys	Move data
CONTROL	Copy data
CONTROL-SHIFT	Link data
ALT	Move data

As with Clipboard data transfers, drag-and-drop transfers involve both a data source and a data destination—a data producer and a data consumer. In both cases, the data producer uses a COleDataSource to originate the data transfer and the data consumer uses a COleDataObject to retrieve the data. In fact, the code for pasting from the Clipboard and that for receiving data in a drag-and-drop operation are usually identical. Many times you can use the same code for retrieving data from a drag-and-drop operation as you would use for a Clipboard paste operation. However, there are a few subtle differences between Clipboard data transfers and drag-and-drop data transfers.

Originating a Drag-and-Drop Transfer

Drag-and-drop data transfers usually begin when the user presses the left mouse button. So, to initiate a drag-and-drop transfer, simply handle the WM_LBUTTONDOWN message.

The mechanics for the data producer are very similar: copy some data into a COleDataSource object using CacheData() or CacheGlobalData(). However, instead of calling COleDataSource::SetClipboard() to set the data to the Clipboard, call COleDataSource::DoDragDrop(), which handles the data transfer automatically. The following snippet demonstrates how to initiate a drag-and-drop operation:

```

COleDataSource cuds;

LPCSTR source = GetString();
HGLOBAL h =
    GlobalAlloc(GHND | GMEM_SHARE, strlen(source) + 1);
strcpy(LPSTR(GlobalLock(h)), source);
GlobalUnlock(h);
cuds.CacheGlobalData(CF_TEXT, h);

DROPEFFECT de =
    cuds.DoDragDrop(DROPEFFECT_COPY | DROPEFFECT_MOVE);
if ( de == DROPEFFECT_MOVE ) {
    // Do any special stuff depending on drop effect
}

```

Notice how DoDragDrop() accepts a combination of DROPEFFECT... values. Those values indicate what kind of transfer operations are allowed. In this example, the program allows copy and move transfer operations.

By calling COleDataSource::DoDragDrop(), you hand off the data to OLE. DoDragDrop() does not return until the user has dropped the data somewhere else (by releasing the left mouse button) or has canceled the transfer by pressing the ESCAPE key.

Implementing a Drop Target

The data consumer of a drag-and-drop operation is called a drop target. Implementing a drop target using MFC is very straightforward. The two classes usually involved in creating one: COleDropTarget and CView. You can also use a CWnd for the target, but you get less default functionality, like no automatic scrolling, for example.

The steps necessary to enable a CView to be a drop target are these: (1) register a COleDropTarget with a CView and (2) override four virtual functions—OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop().

To register a view as a drop target, declare a member variable of type COleDropTarget in your view class. Then handle the view's OnCreate() method (called right after a view is created). Inside OnCreate(), call COleDropTarget::Register(), sending it the view's this pointer. Underneath the hood, COleDropTarget() calls the RegisterDragDrop() function, which registers a window handle as a drag-and-drop target within OLE.

Once a window is registered with OLE to be a drop target, it becomes eligible to receive notifications that correspond to CView's OnDragEnter(), OnDragOver(), OnDragLeave(), OnDrop() functions. These functions are important because they allow the user interface to be updated during the drag-and-drop operation.

Drag-and-drop transfers involve changes in the user interface. The normal feedback for drag-and-drop operations is represented by one of several standard cursors. If a

window cannot accept a data drop, then the program should display a circle with a slash through it to show that the window is not a drop target. If the window can accept a drop and the data is being moved (as opposed to being copied), then the cursor should appear as the regular arrow cursor but with a small square (representing data) to the lower right of the arrow. If the window can accept a drop and the data is being copied, then the program should show the same cursor it does when data is being moved, except that there should be a small plus sign to the upper right of the arrow. Fortunately, MFC handles the cursor changes very easily through the return values of the drag-and-drop handler functions. Here's how the four drag-and-drop notification functions work.

Whenever the mouse cursor enters a window registered as a drop target, the framework calls `OnDragEnter()`. The framework calls the view's `OnDragOver()` function as the mouse cursor is dragged over a drop target.

`OnDragEnter()` and `OnDragOver()` are very closely related. Both accept three parameters: a pointer to a `COleDataObject`, a `DWORD` indicating the state of the control keys, and a `CPoint` object indicating the location of the mouse. Both functions return a `DROPEFFECT...` value, which indicates the type of data transfer occurring. There are five different `DROPEFFECT...` values, which indicate the following drag-and-drop transfer states:

- `DROPEFFECT_NONE`—A drop is not allowed.
- `DROPEFFECT_COPY`—A copy operation is pending.
- `DROPEFFECT_MOVE`—A move operation is pending.
- `DROPEFFECT_LINK`—A link from the dropped data to the original data is pending.
- `DROPEFFECT_SCROLL`—A drag/scroll operation is about to occur or is occurring in the target.

`OnDragEnter()` and `OnDragOver()` can use the information in the `COleDataObject`, the key press mask, and the `CPoint` parameters to determine what kind of operation is being performed. For example, if the drop target accepts only `CF_BITMAP`-formatted data and the Clipboard contains only `CF_TEXT`-formatted data (which you can find out by querying the `COleDataObject`), the view cannot accept the data. In this case, `OnDragEnter()` and `OnDragOver()` would return `DROPEFFECT_NONE` to indicate that the view cannot accept a drop. By returning the appropriate `DROPEFFECT...` value, these functions tell the framework which cursor to display.

In addition, the `OnDragEnter()` and `OnDragOver()` functions can check the key mask (the `DWORD` parameter) to find out if any control keys are pressed. To change the cursor, the drop target need only return the appropriate `DROPEFFECT...` value and MFC does the rest.

The view's `OnDragLeave()` function is called when the mouse cursor leaves the view. It takes no parameters and returns void. This is a good place to put any needed drag-and-drop clean-up code.

The final function, `OnDrop()`, is called whenever the user releases the left mouse button inside a view that is a drop target. `OnDrop()` takes the same parameters as `OnDragEnter()` and `OnDragOver()`: a pointer to a `COleDataObject`, a `DWORD` indicating the state of the control keys, and a `CPoint` object indicating the location of the mouse. `OnDrop()` also returns a `DROPEFFECT...` value that indicates the type of transfer operation that actually occurred. This is the value passed back to `COleDataSource::DoDragDrop()`. Again, the code for dropping data and that for pasting it from the Clipboard are virtually identical. Many developers decide to implement one body of code for both pasting from the Clipboard and accepting a data drop.

Inside MFC's Drag-and-Drop Classes

In summary, MFC's drag-and-drop data transfers are really an extension of the Clipboard transfer. As with Clipboard transfers, drag-and-drop transfers start off with the `COleDataSource`. A data producer either caches or promises a rendering later (using `DelayRenderData()`). However, instead of calling `SetClipboard()`, the data producer calls `DoDragDrop()`. On the data consumer side, the application registers one of its views with a `COleDropTarget`. Then the view receives drag-and-drop notifications. Here's a closer look at MFC's drag-and-drop-related classes.

`COleDataSource`

In addition to holding a data cache and providing support for Clipboard transfers, `COleDataSource` has a function called `DoDragDrop()`. `DoDragDrop()` wraps the OLE function `DoDragDrop()`, which has this prototype:

```
DROPEFFECT DoDragDrop(
    DWORD dwEffects = DROPEFFECT_COPY|DROPEFFECT_MOVE|DROPEFFECT_LINK,
    LPCRECT lpRectStartDrag = NULL,
    COleDropSource* pDropSource = NULL);
```

Notice that `COleDataSource::DoDragDrop()` takes three parameters: (1) a `DROPEFFECT` indicating which drag-and-drop operations are allowed, (2) a rectangle that determines when the drag-and-drop operation actually starts, and (3) a pointer to a `COleDropSource` that handles the user interface aspects of the drag-and-drop operation. If the last parameter is `NULL`, `DoDragDrop()` uses a standard `COleDropSource` implementation already provided by MFC.

COleDropSource

COleDropSource implements the IDropSource interface. Listing 12-4 shows the definition of COleDropSource.

Listing 12-4. The COleDropSource class

```
class COleDropSource : public CCmdTarget {
// Constructors
public:
    COleDropSource();

// Overridables
    virtual SCODE QueryContinueDrag(BOOL bEscapePressed, DWORD dwKeyState);
    virtual SCODE GiveFeedback(DROPEFFECT dropEffect);
    virtual BOOL OnBeginDrag(CWnd* pWnd);

public:
    BEGIN_INTERFACE_PART(DropSource, IDropSource)
        INIT_INTERFACE_PART(COleDropSource, DropSource)
        STDMETHOD(QueryContinueDrag)(BOOL, DWORD);
        STDMETHOD(GiveFeedback)(DWORD);
    END_INTERFACE_PART(DropSource)

    DECLARE_INTERFACE_MAP()

    CRect m_rectStartDrag;
    BOOL m_bDragStarted;
    DWORD m_dwButtonCancel;
    DWORD m_dwButtonDrop;

    // metrics for drag start determination
    static AFX_DATA UINT nDragMinDist;
    static AFX_DATA UINT nDragDelay;

    friend class COleDataSource;
};


```

COleDropSource implements the IDropSource interface via the normal MFC interface maps. IDropSource::QueryContinueDrag() and IDropSource::GiveFeedback() map directly to the virtual functions COleDropSource::QueryContinueDrag() and COleDropSource::GiveFeedback(). Because these functions are virtual, you can safely override them.

There are several reasons you might want to override MFC's default implementation of COleDropSource. If you're unhappy with the default cursors provided by MFC, all you need to do is derive a class from COleDropSource and override the GiveFeedback() function. In addition, COleDropSource maintains variables for identifying the key for canceling the drag-and-drop operation (m_dwButtonCancel) and

for specifying the mouse button that identifies the drop operation (*m_dwButtonDrop*).

COleDropSource also has static members for specifying the time delay before the drag-and-drop operation starts (*m_nDragDelay*) and for specifying the number of pixels the mouse must move before the drag-and-drop operation starts. COleDropSource's constructor initializes these variables. The drag distance and drag delay variables are read in from the WIN.INI file. The drag distance's key is "DragMinDist" and the drag delay's key is "DragDelay". If these entries aren't in the initialization file, then the drag distance's value is set to 2 and the drag delay's set to 200 milliseconds.

To change any of the user-interface characteristics for a given drag-and-drop operation, just derive your own class from COleDropSource. Then feed your newly derived class into COleDataSource::DoDragDrop().

COleDropTarget

COleDropTarget implements the IDropTarget interface. COleDropTarget is a real COM object implemented using MFC's interface maps. Listing 12-5 shows its declaration.

Listing 12-5. The COleDropTarget class

```
class COleDropTarget : public CCmdTarget {
// Constructors
public:
    COleDropTarget();

// Operations
    BOOL Register(CWnd* pWnd);
    virtual void Revoke(); // virtual for implementation

// Overridables
    virtual DROPEFFECT OnDragEnter(CWnd* pWnd, COleDataObject* pDataObject,
        DWORD dwKeyState, CPoint point);
    virtual DROPEFFECT OnDragOver(CWnd* pWnd, COleDataObject* pDataObject,
        DWORD dwKeyState, CPoint point);
    virtual BOOL OnDrop(CWnd* pWnd, COleDataObject* pDataObject,
        DROPEFFECT dropEffect, CPoint point);
    virtual DROPEFFECT OnDropEx(CWnd* pWnd, COleDataObject* pDataObject,
        DROPEFFECT dropDefault, DROPEFFECT dropList, CPoint point);
    virtual void OnDragLeave(CWnd* pWnd);
    virtual DROPEFFECT OnDragScroll(CWnd* pWnd, DWORD dwKeyState,
        CPoint point);

// Implementation
public:
    virtual ~COleDropTarget();

protected:
    HWND m_hWnd;                                // HWND this IDropTarget is attached to
```

```

LPDATAOBJECT m_lpDataObject; // != NULL between OnDragEnter, OnDragLeave
UINT m_nTimerID;           // != MAKEWORD(-1, -1) when in scroll area
DWORD m_dwLastTick;        // only valid when m_nTimerID valid
UINT m_nScrollDelay;       // time to next scroll

// metrics for drag-scrolling
static AFX_DATA int nScrollInset;
static AFX_DATA UINT nScrollDelay;
static AFX_DATA UINT nScrollInterval;

// implementation helpers
void SetupTimer(CView* pView, UINT nTimerID);
void CancelTimer(CWnd* pWnd);

// Interface Maps
public:
BEGIN_INTERFACE_PART(DropTarget, IDropTarget)
    INIT_INTERFACE_PART(COleDropTarget, DropTarget)
    STDMETHOD(DragEnter)(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
    STDMETHOD(DragOver)(DWORD, POINTL, LPDWORD);
    STDMETHOD(DragLeave)();
    STDMETHOD(Drop)(LPDATAOBJECT, DWORD, POINTL pt, LPDWORD);
END_INTERFACE_PART(DropTarget)

DECLARE_INTERFACE_MAP()
};

COleDropTarget's most important members are Register(), DragEnter(), DragOver(), DragLeave(), and Drop().

```

The other important class in MFC's drag-and-drop implementation is CView. COleDropTarget works in concert with CView to make it very easy to program OLE drag-and-drop.

CView

You may remember that CView also has OnDragEnter(), OnDragLeave(), OnDragOver(), and OnDrop() functions. But wait—these are normally handled by IDropTarget. What are they doing in CView? How are these functions ever called? Let's follow MFC's drag-and-drop operation from start to finish to see exactly what happens.

How MFC Drag-and-Drop Works

The best way to see how MFC does OLE drag-and-drop is to examine each side separately: the data producer side and the data consumer side. On the producer side, there's COleDataSource and COleDropSource. On the data consumer side, there's COleDataObject and COleDropTarget. Let's examine the data producer side first.

The Data Producer

To be a drag source, a data producer has only to either cache the data (or use DelayRenderData() to set up delayed rendering) and then call DoDragDrop().

At the surface, COleDataSource::DoDragDrop() appears to be more or less a wrapper for the API function ::DoDragDrop(). However, COleDataSource::DoDragDrop() does a bit more than just act as a wrapper for ::DoDragDrop().

When the data producer calls COleDataSource::DoDragDrop(), MFC examines the COleDropSource passed in by the caller. If the caller has provided a COleDropSource, DoDragDrop() uses that COleDropSource to manage the user interface (that is, change the cursors and determine under which conditions the drag-and-drop operation can continue or is canceled). If the caller doesn't provide a COleDropSource, DoDragDrop() uses the default implementation of COleDropSource provided by MFC.

If the caller provides a rectangle parameter, DoDragDrop() copies that rectangle into its own member rectangle (called m_rectStartDrag). COleDataSource uses this rectangle to know when to start dragging. That is, the actual dragging won't commence until the cursor leaves this rectangle. If the data producer passes a NULL pointer, DoDragDrop() creates its own empty rectangle around the current position of the cursor.

Before starting the drag operation, COleDataSource::DoDragDrop() calls OnBeginDrag(). This function is virtual, so you can override it if you'd like. The default implementation of OnBeginDrag() first captures mouse input and then sets up a loop that acts as a timer. OnBeginDrag() watches the tick count go down.

Then COleDataSource::DoDragDrop() uses GetInterface() to retrieve the IDataObject and the IDropSource interfaces from the COleDropSource that was passed in. COleDataSource::DoDragDrop() uses these interface pointers to call the API function ::DoDragDrop().

Earlier versions of MFC called DoDragDrop() directly. However, now MFC maintains a list of pointers to functions that implement OLE-specific features. For example, here's the code MFC uses to call DoDragDrop():

```
#ifdef _AFXDLL
    _afxOLE.pfnDoDragDrop[0](lpDataObject, lpDropSource, dwEffects,
                           &dwResultEffect);
#else
    ::DoDragDrop(lpDataObject, lpDropSource, dwEffects, &dwResultEffect);
#endif
```

In MFC4, Microsoft combined all the separate MFC DLLs into one monolithic DLL. If the applications linked directly to various DLLs (like OLE32.DLL) the application would load OLE even for very simple apps. For this reason, Microsoft loads all non-essential DLLs dynamically using a very clever technique. Most of these instances

are hidden by macros, except where not possible, as in this case. MFC40.DLL only binds directly to KERNEL32.DLL, GDI32.DLL, USER32.DLL, and MSVCRT20.DLL. All other DLLs are loaded at runtime when the first function in them is called. OLE32.DLL, OLEAUT32.DLL, SHELL32.DLL, ADVAPI32.DLL, COMCTL32.DLL, ODBC32.DLL, WINSPOOL.DLL, are all examples of DLLs that are loaded at runtime. This allows MFC to maintain one giant DLL but with no performance hit for applications that don't use all these DLLs.

The Data Consumer

Again, like Clipboard data transfers, drag-and-drop operations culminate in the production of a COleDataObject. However, instead of pulling an IDataObject pointer off the Clipboard using OleGetClipboard(), MFC provides a fully loaded COleDataObject whenever data is dropped on a view.

Registering with OLE

To receive drag-and-drop notifications, a view must first register itself with a drop target. That is, the view must declare an instance of COleDropTarget and call the Register() function. The standard place for calling Register() is during the view's OnCreate() function, because that's the first place you can provide any initialization for a window. The following code snippet shows how to do this:

```
class CMyView: public CView {
...
    COleDropTarget m_dt;
...
};

CMyView::OnCreate() {
    m_dt.Register(this);
}
```

So just what does Register() do? Before doing anything, Register() calls CoLockExternalObject() to place a strong lock on the drop target object. That way, the drop target object is guaranteed to remain in memory even if the data consumer calls Release() enough times to balance the AddRef()'s. If CoLockExternalObject() weren't called, the drag-and-drop operation would work once but fail every time thereafter.

Next, Register() calls the OLE API function RegisterDragDrop(). The API function takes two parameters: (1) the handle of the window that is to receive drag-and-drop notifications and (2) a pointer to an IDropTarget interface. Once this information is registered with OLE using RegisterDragDrop(), then OLE will be able to call the

IDropTarget functions (that is, OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop()) as the mouse cursor moves in and around the window specified by the window handle. Before leaving, COleDropTarget::Register() saves both the CWnd object and the window handle passed to Register(). COleDropTarget uses the CWnd object to implement OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop(). COleDropTarget needs the window handle to implement its Revoke() member function.

Finally, CWnd has a pointer to a COleDropTarget member. In addition to saving the CWnd object and the window handle, COleDropTarget::Register() sets the CWnd's COleDropTarget member to point to the COleDropTarget's this pointer. CWnd uses the pointer to the drop target to revoke drag-and-drop registration when the window is destroyed.

Revoking Drag-and-Drop Registration

COleDropTarget contains a complementary function that revokes a window's drag-and-drop registration. The function is called Revoke() (surprisingly enough!). COleDropTarget::Revoke() calls the OLE API function RevokeDragDrop() and frees the external lock on the COleDropTarget. As a developer, you'll rarely find yourself calling this function. As part of its OnNcDestroy() handler, CWnd calls COleDropTarget::Revoke() if the CWnd has a pointer to the COleDropTarget.

Handling the Notification

Once a window is registered with OLE to receive drag-and-drop notifications, it'll start receiving those messages as soon as the mouse cursor enters the boundaries of a window. In the case of MFC, the drag-and-drop notifications are handled by the CView class. So, though you can pass a plain-vanilla CWnd when registering a window to be a drop target, it's the CView that already has handlers for OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop(). MFC's drag-and-drop won't work if you register the drop target with something other than a CView.

Whenever the mouse cursor passes over a window that is registered to receive drag-and-drop notifications, OLE starts sending notifications to the drop target. The first message called during dragging is OnDragEnter(). Remember, COleDropTarget implements IDropTarget. If the window and the IDropTarget pointer are registered with OLE, OLE calls IDropTarget::DragEnter(). When OLE calls the IDropTarget's DragEnter() function, OLE passes the fully loaded IDataObject (the one provided by the data producer) and the window handle of the window over which the cursor was dragged. COleDropTarget's implementation of DragEnter() attaches the loaded IDataObject to a COleDataObject and looks up the window handle by calling CWnd::FromHandle(). COleDropTarget's DragEnter() simply delegates to the OnDragEnter() member function passing the CWnd and the COleDataObject. Inside

COleDropTarget::OnDragEnter() checks to see if the window attached is a view. If the attached window isn't a view, then OnDragEnter() exits, returning DROPEFFECT_NONE. Otherwise, COleDropTarget::OnDragEnter() tries to cast the CWnd pointer to a CView. If the cast is successful, COleDropTarget::OnDragEnter() delegates to CView::OnDragEnter(). At that point, the target view can examine the data, determine if it's useful, and update the user interface accordingly.

The other IDropTarget functions work in a similar way. When OLE calls the IDropTarget interface function represented by COleDropTarget, COleDropTarget takes the IDataObject pointer, attaches it to a COleDataObject, finds the CWnd object representing the window, casts it to a view, and calls the corresponding CView function (that is, OnDragOver(), OnDragLeave(), or OnDrop()).

As you can see, a lot of work goes into implementing drag-and-drop in a safe, robust way. Fortunately, MFC's implementation does most of the work for you.

Conclusion

Uniform Data Transfer in OLE is simpler than you might expect—especially when you use the MFC wrappers for IDataObject. Programming data transfers using the Microsoft Foundation Classes means that you can cut down your exposure to the OLE gunk (the low-level API). The information provided in this chapter gives you a good start in understanding how to use UDT to implement Clipboard and drag-and-drop data transfer techniques. In the next chapter, we cover OLE documents. For more information, be sure to check the DNEDIT.EXE example on the diskette. DNEDIT shows how to use UDT within the context of a simple Notepad-type text editor.

OLE Documents the MFC Way

And in the beginning there was Object Linking and Embedding.

Those of us Windows enthusiasts old enough to remember late 1991 and early 1992 may recall the hoopla surrounding Microsoft's new extension to dynamic data exchange called Object Linking and Embedding. During their road shows, Microsoft salesmen would express with great zeal the ability to plant Microsoft Excel spreadsheets and Microsoft Paint pictures within a Word for Windows document. The presenters would deftly double-click on the picture embedded within the document and Paintbrush magically fired up in another window! For the early 1990s, this sort of stuff was simply amazing.

Microsoft aptly named this new technology Object Linking and Embedding because users could link and embed objects from various sources into a single document. Now that there's a newer OLE, based on the Component Object Model, the original version of OLE is called OLE1. OLE1 was a pretty cool way to mix different kinds of data in a single document. For creating reports from various data sources, OLE1 sure beat messing with importing and exporting files, getting paper cuts handling printouts, and risking stapling your tie to the report. However, because it was really just a new wrapper around DDE, it had all the same problems as DDE: it was asynchronous, it was slow at times, and it was inflexible because the only data transfer media it worked with were global memory handles.

These days, OLE based on COM (the real OLE) is much more flexible than OLE1 (based on DDE). As you've seen in earlier chapters, OLE and COM support many features. The compound document feature (now called OLE documents) is just one aspect of an ever-widening set of technologies being defined by OLE. This chapter covers OLE documents and how MFC implements them. Along the way, we'll answer such burning questions as these:

- How does MFC manage compound files? How is OLE's Structured Storage model integrated into MFC's serialization model?

- What happens when a user double-clicks on an in-place item? How is all that menu hocus-pocus accomplished?
- How do some of the raw OLE interface functions map to various MFC class member functions?

OLE Documents 101

If you haven't seen OLE documents in action yet, give one a try. They're quite something to behold. Go ahead and embed an Excel spreadsheet in your Word document. There in the middle of your text lies a spreadsheet. Double-click on the spreadsheet and suddenly you're looking at Excel. Click outside the spreadsheet and you're looking at Word again. What they can't do these days!

The idea behind OLE documents is to collect lots of different data in one place. That is, applications that follow the OLE document model mix their native data with data from other sources. The classic example is mixing some spreadsheet data into a word processor document. The word processor then manages its own data as well as the spreadsheet data.

The OLE document model also specifies that the user be able to edit the content of the spreadsheet data (in our example) by double-clicking on the spreadsheet representation in the word processor.

Though being able to do all this is extremely convenient for the user, it's a fairly daunting programming task. As you can imagine, many details require your attention. For example, the spreadsheet data is probably completely different from the word processor data. How do you mix the two types of data in one file? In addition, how do you magically transform one application into another (as when you double-click on the spreadsheet and the Excel interface appears)? There's a lot of work involved in doing this stuff. This is one area where MFC does more than its share of the work.

However, before digging deep into OLE documents, let's learn a few new terms and concepts: linking, embedding, the Structured Storage model, in-place activation, and visual editing.

Linking and Embedding

In OLE, there are two ways to integrate different forms of data in a single document: linking and embedding. Linked documents and embedded documents are distinguished by where the data is stored.

When data is linked in an OLE document, that data is stored in a separate file and the OLE document contains reference to that data. When data is embedded in an OLE document application, the OLE document stores the embedded data right alongside the application's native data in a single file.

For example, if you take an Excel spreadsheet file and put it inside your Word for Windows document, you have a choice of linking or embedding the item. If you link the Excel spreadsheet, then the data remains stored in the Excel file (outside of the Word document). If you embed the Excel spreadsheet, then the data is stored inside the Word file, right along with the native Word data.

Figures 13-1 and 13-2 illustrate the difference between linking and embedding.

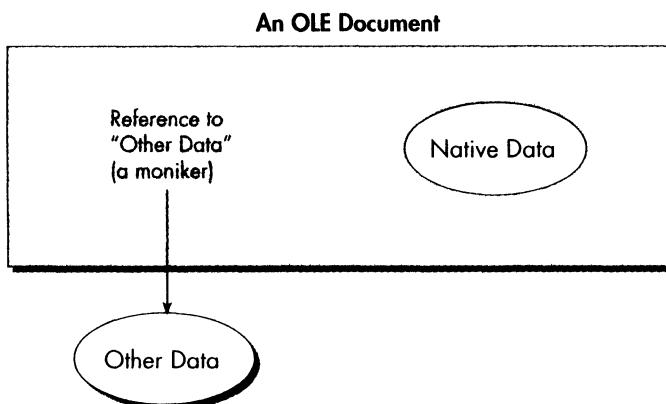


Figure 13-1. An OLE document containing a linked item

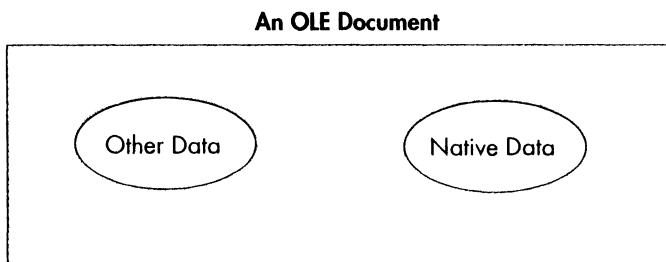


Figure 13-2. An OLE document containing an embedded item

Linking versus Embedding

Obviously, a document containing a link to data (as opposed to a separate copy) will tend to be considerably smaller than a document containing the same data as an embedding. However, one disadvantage of linking data is that moving the data

source behind a linked item will break the link. In addition, if you want to distribute a linked OLE document, you need to distribute both parts: the OLE document *and* the file containing the linked data. If you embed the data, then you need distribute only the OLE document. Because the nonnative data is actually stored in a single file, that's the only file the users of the OLE document need.

Of course, integrating many different forms of data in a single document presents another problem. That is, how do you manage storing different types of data in a single file? The answer is OLE's Structured Storage model.

Structured Storage, Compound Files, and Persistent Objects

If you were to try to solve this problem in the Dark Ages (that is, pre-OLE), you might do something akin to defining some sort of file structure that includes header information pointing to various locations in the file containing the actual information. People actually used to create file formats like that! *Arrgh*.

There are a couple of problems with this approach. First, it's a really gnarly programming task. There's a whole lot to keep track of when doing something like this. Keeping the file in sync would likely be a nightmare. The other major problem with this approach is that it ties the file format to the application. Any other application that wants to read that data has to understand the file format. You can probably see by this point that some sort of uniform storage model is needed.

OLE provides such a model: it's called Structured Storage. Microsoft even provides an implementation of the Structured Storage model, called compound files.

Structured Storage

Here's a good way to think about Structured Storage. Structured Storage treats the structure of a file very much the way the operating system treats the directory structure. You've probably heard the description before: it's a file system within a file. Just as the file system maintains a hierarchy of directories and files, an OLE Structured Storage file maintains a similar hierarchical structure using *storages* and *streams*. In the Structured Storage model, storages are analogous to directories and streams are analogous to files. Structured Storage organizes a file around storages and manages actual data in streams.

Figure 13-3 shows a screen shot from a program called DFVIEW.EXE. (DFVIEW comes with Visual C++. It is a compound file browser.) Notice how the root storage has separate substorages hanging off of it, and how the substorages have streams beneath them. This should give you a good idea how Structured Storage works.

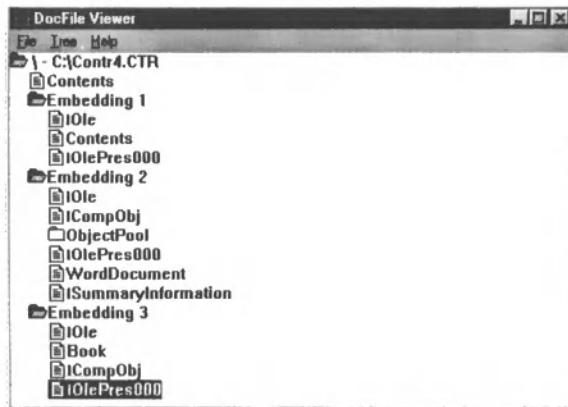


Figure 13-3. DFVIEW.EXE showing the contents of a compound document file

As with all things OLE, Structured Storage revolves around some interfaces. Three interfaces in particular define the Structured Storage model: IStorage, IStream, and ILockBytes. IStorage has functions for managing storage organization, IStream has functions for handling reading and writing data, and ILockBytes handles the low-level functionality necessary for abstracting a storage device so that it appears to the client as an array of bytes.

It's not necessary to understand these interfaces completely to see how MFC implements OLE documents. However, it's important to get the gist of each interface. Let's look at the IStorage interface first, as shown in Listing 13-1.

Listing 13-1. The IStorage interface

```
interface IStorage : public IUnknown {
public:
    virtual HRESULT CreateStream(const OLECHAR FAR *pwcsName,
        DWORD grfMode,
        DWORD reserved1,
        DWORD reserved2,
        IStream FAR *FAR *ppstm) = 0;
    virtual HRESULT OpenStream(const OLECHAR FAR *pwcsName,
        void FAR *reserved1,
        DWORD grfMode,
        DWORD reserved2,
        IStream FAR * FAR *ppstm) = 0;
    virtual HRESULT CreateStorage(const OLECHAR FAR *pwcsName,
        DWORD grfMode,
        DWORD dwStgFmt,
        DWORD reserved2,
        IStorage FAR *FAR *ppstg) = 0;
```

```

virtual HRESULT OpenStorage(const OLECHAR FAR *pwcsName,
                           IStorage FAR *pstgPriority,
                           DWORD grfMode,
                           SNB snbExclude,
                           DWORD reserved,
                           IStorage FAR * FAR *ppstg) = 0;
virtual HRESULT CopyTo(DWORD ciidExclude,
                      const IID FAR *rgiidExclude,
                      SNB snbExclude,
                      IStorage FAR *pstgDest) = 0;
virtual HRESULT MoveElementTo(const OLECHAR FAR *pwcsName,
                           IStorage FAR *pstgDest,
                           const OLECHAR FAR *pwcsNewName,
                           DWORD grfFlags) = 0;
virtual HRESULT Commit(DWORD grfCommitFlags) = 0;
virtual HRESULT Revert( void ) = 0;
virtual HRESULT EnumElements(DWORD reserved1,
                            void FAR *reserved2,
                            DWORD reserved3,
                            IEnumSTATSTG FAR * FAR *ppenum) = 0;
virtual HRESULT DestroyElement(const OLECHAR FAR *pwcsName) = 0;
virtual HRESULT RenameElement(const OLECHAR FAR *pwcsOldName,
                           const OLECHAR FAR *pwcsNewName) = 0;
virtual HRESULT SetElementTimes(const OLECHAR FAR *pwcsName,
                           const FILETIME FAR *pctime,
                           const FILETIME FAR *patime,
                           const FILETIME FAR *pmtime) = 0;
virtual HRESULT SetClass(REFCLSID clsid) = 0;
virtual HRESULT SetStateBits(DWORD grfStateBits,
                           DWORD grfMask) = 0;
virtual HRESULT Stat(STATSTG FAR *pststatstg,
                     DWORD grfStatFlag) = 0;
};

}

```

IStorage is just like the other interfaces you've already encountered. It's a pure virtual base class with functions just waiting to be implemented. The important thing to notice here is the kind of functions included in the interface. All the functions deal with managing and organizing storage. This is especially evident with functions such as CreateStorage() and CreateStream().

Now take a look at the IStream interface, in Listing 13-2.

Listing 13-2. The IStream interface

```

interface IStream : public IUnknown {
public:
    virtual HRESULT Read(void FAR *pv,
                        ULONG cb,
                        ULONG FAR *pcbRead) = 0;

```

```

virtual HRESULT Write(const void FAR *pv,
                     ULONG cb,
                     ULONG FAR *pcbWritten) = 0;
virtual HRESULT Seek(LARGE_INTEGER dlibMove,
                     DWORD dwOrigin,
                     LARGE_INTEGER FAR *plibNewPosition) = 0;
virtual HRESULT SetSize(ULARGE_INTEGER libNewSize) = 0;
virtual HRESULT CopyTo(IStream FAR *pstm,
                     ULONG cb,
                     ULONG FAR *pcbRead,
                     ULONG FAR *pcbWritten) = 0;
virtual HRESULT Commit(DWORD grfCommitFlags) = 0;
virtual HRESULT Revert( void) = 0;
virtual HRESULT LockRegion(ULARGE_INTEGER libOffset,
                     ULONG cb,
                     DWORD dwLockType) = 0;
virtual HRESULT UnlockRegion(ULARGE_INTEGER libOffset,
                     ULONG cb,
                     DWORD dwLockType) = 0;
virtual HRESULT Stat(STATSTG FAR *pstatstg,
                     DWORD grfStatFlag) = 0;
virtual HRESULT Clone(IStream FAR *FAR *ppstm) = 0;
};

}

```

IStream has functions necessary for reading and writing data to and from some device: notice functions such as Read(), Write(), and Seek(). IStream's functions are similar to the run-time library functions you might use to read and write to a regular file.

Finally, take a moment to examine ILockBytes, in Listing 13-3.

Listing 13-3. The ILockBytes interface

```

interface ILockBytes : public IUnknown {
public:
    virtual HRESULT ReadAt(ULARGE_INTEGER ulOffset,
                          void FAR *pv,
                          ULONG cb,
                          ULONG FAR *pcbRead) = 0;
    virtual HRESULT WriteAt(ULARGE_INTEGER ulOffset,
                          const void FAR *pv,
                          ULONG cb,
                          ULONG FAR *pcbWritten) = 0;
    virtual HRESULT Flush( void) = 0;
    virtual HRESULT SetSize(ULARGE_INTEGER cb) = 0;
    virtual HRESULT LockRegion(ULARGE_INTEGER libOffset,
                          ULONG cb,
                          DWORD dwLockType) = 0;
}

```

```

virtual HRESULT UnlockRegion(ULARGE_INTEGER libOffset,
                            ULARGE_INTEGER cb,
                            DWORD dwLockType) = 0;
virtual HRESULT Stat(STATSTG FAR *pstatstg,
                     DWORD grfStatFlag) = 0;
};

}

```

The point of implementing a system like Structured Storage is to provide one set of interfaces that'll work over a variety of operating systems. This is where **ILockBytes** comes in. **ILockBytes** has the functions necessary for treating a storage device like an array of bytes. Clients using **ILockBytes** can view a storage medium as an array of bytes. Consider DOS's file system: the File Allocation Table system. When you call C run-time functions for manipulating data, you can remain blissfully ignorant of the complexities involved in managing disk sectors and clusters. The operating system takes care of all that for you. By the same token, OLE objects that implement **ILockBytes** manage the same sorts of complexities so that **ILockBytes** clients can deal with a byte array.

As you can see, these are three nontrivial interfaces. For now, the most important thing is that you know they exist and understand how they are intended to be used. Again, **IStorage** has functions for managing storages, **IStream** has functions for performing reading and writing operations on streams, and **ILockBytes** behaves as a sort of "device driver," abstracting a storage device to a flat byte array.

Compound Files

As a programmer, you probably don't want to have to implement those interfaces. After all, this Structured Storage model is really the domain of the operating system. (We know that it's something we'd like to avoid implementing, if at all possible.) Fortunately, Structured Storage is one of the features for which Microsoft provides an implementation: compound files.

Consider the way you do file I/O in a regular program. If you want to create a file, you usually call a file-related function such as `_creat()` or `_open()` from the run-time library. The file-management functions represent files using a file handle. The file handle is a magic cookie used by the operating system to manage disk and file I/O. For example, successfully calling `_creat()` or `_open()` returns a valid file handle that you can use in subsequent file operations. Once you acquire a file handle, you can use functions such as `_lseek()`, `_read()`, and `_write()` to navigate through the file and access its contents.

Microsoft's implementation of Structured Storage, compound files, works a little bit differently. Instead of calling a run-time library function to create or open a file, you call a global API function. The most commonly used function for creating a compound file storage is `StgCreateDocFile()`. However, instead of yielding a file

handle, StgCreateDocFile() hands you (ten dollars to the reader who guesses this first) an interface pointer. Specifically, you get back an IStorage interface. Once the operating system hands back an IStorage pointer, you can use the interface to manage storages and streams.

For example, given an IStorage pointer, you can create a new substorage by calling IStorage::CreateStorage(). Calling IStorage::CreateStream() creates a new stream under the storage. You may notice that IStorage doesn't have a Close() function. Remember, IStorage is just another OLE interface. Calling Release() closes the storage.

Notice that calling IStorage::CreateStream() gives you back an IStream pointer. You can use the IStream pointer to read and write to a stream. In short, you can treat it almost as you would a normal file.

Of course, you're not confined to using Microsoft's Structured Storage model compound files. IStorage, IStream, and ILockBytes are just OLE interfaces awaiting implementation. But Microsoft's compound files work pretty well: there aren't many good reasons to stray from it. In fact, MFC uses compound files to provide storage for OLE documents.

So, the IStorage, IStream, and ILockBytes interfaces compose one side of the storage equation. COM objects obtain IStorage and IStream interfaces and write data to some persistent medium. But how do you create a persistent object? Telling COM objects to save themselves is the other side of the object persistence story. OLE defines another set of interfaces for that purpose.

Persistent Objects

Persistent objects know how to save themselves to some medium (such as a disk). In OLE, persistent objects expose one of the persistence interfaces: IPersistStorage, IPersistStream, or IPersistFile.

By exposing IPersistStorage, COM objects advertise that they can store themselves to an OLE Structured Storage object. IPersistStorage consists of such functions as Load(), Save(), and IsDirty(). Clients use the functions in this interface to tell COM objects to save themselves using an OLE Structured Storage object. COM objects expose IPersistStream to tell their clients that the objects can persist to an OLE Structured Storage stream. Finally, IPersistFile tells COM object clients that the object can save itself to a regular old disk file (as opposed to an OLE storage or stream).

Next, let's take a look at in-place activation and visual editing.

In-Place Activation and Visual Editing

An OLE document holds OLE document *content objects*. In our example, the spreadsheet is a content object. When content objects are part of an OLE document, OLE calls them *in-place* items. In an OLE document, these items can often be activated

and edited in place. That is, the user can edit the content without being forced to switch applications to edit the object. This feature—the ability of one application to take over the user interface of another—and the ability to edit the content object without leaving the host application is called *in-place activation*.

In-place activation means telling a document item to do something through one of its *verbs*. Linked and embedded objects can support a number of different verbs. The most common ones are Open and Edit. When executing the Open verb, an OLE document in-place item usually opens the server application in another window. When executing the Edit verb, an OLE document in-place object replaces the container's main menu bar with a composite menu bar comprising elements from both the container application's and the server application's menu bars.

In addition, an OLE document object may include other verbs as well. For example, the Microsoft Media Player includes a Play verb, which tells the Media Player to play an AVI file.

Of course, as with any client/server situation, there are two ends to the OLE document equation: containers and servers. Containers hold content objects supplied by OLE document servers.

OLE Document Containers

Container applications host OLE documents. That is, they hold OLE document content objects. In our example, the word processor application is the container and the spreadsheet application is the server. A container application is responsible for the following duties within the OLE document equation:

- Providing storage for the entire document.
- Keeping an ear turned toward the content object to intercept change notifications.
- Responding to mouse double-clicks and verb menus.

To perform these responsibilities, containers implement the IOleClientSite, IAdviseSink, and IOleInplaceSite interfaces, at the bare minimum.

Containers implement IOleClientSite to give the content object a way to communicate with and request services from its container. You can also think of IOleClientSite as the anchor point for the content object in the OLE document. The container provides one IOleClientSite instance per content object. Don't worry, we'll look at the interface in more depth soon.

Containers also implement IAdviseSink. OLE document content objects use the container's IAdviseSink interface to receive notifications of data changes, view changes, and compound-document changes. For example, container applications require such notifications to keep the presentations of their linked and embedded

objects up-to-date. OLE document container applications plug an instance of the `IAdviseSink` interface into the content object's `IDataObject` interface (that's what `IDataObject::DAdvise()` is for). Once the content object has the `IAdviseSink` pointer, it calls the interface member functions to notify the container of various changes. As with `IOleClientSite`, we'll look at `IAdviseSink` in more depth soon.

Finally, `IOleInPlaceSite` manages the user-interface side of an OLE document container whenever the server comes in and takes over the container's menus.

OLE document servers provide the other side of the equation.

OLE Document Servers

Whereas OLE document containers host content objects, OLE document servers provide the content objects. OLE document servers have the following responsibilities:

- Storing the specific data in a storage provided by the container.
- Notifying the container of changes (using the container's `IAdviseSink` interface).
- Providing in-place resources (menus and toolbars) if the content object can be edited in place
- Assisting with the menu/user-interface negotiating.

OLE document servers implement at least three interfaces: `IOleObject`, `IPersistStorage`, and `IDataObject`.

Let's look at `IOleObject` first. This is another doozy of an interface, with 21 (count 'em) functions. The container uses this interface as the main communication link to the content object. For example, `IOleObject` contains such functions as `DoVerb()` and `GetExtent()`. Whenever a container wants to tell a content object to do something, the container calls the content object's `IOleObject::DoVerb()` function. Whenever the container wants to find out how big the content object is, the container calls the content object's `IOleObject::GetExtent()`. MFC implements this interface in its `COleServerItem` and `COleServerDoc` classes.

You've already seen `IDataObject` in the Uniform Data Transfer function. OLE document objects expose this interface so that containers can retrieve presentation information from the content object.

OLE document content objects implement `IPersistStorage` so that containers can tell them to persist. Notice that `IPersistStorage` has functions such as `Load()` and `Save()`. OLE document containers acquire these interfaces from content objects and use the functions to load and save OLE document content objects.

These are the most important interfaces necessary for implementing OLE documents. They're not the only ones by any means. We'll see more interfaces for dealing with in-place activation as we dig down into MFC's OLE document support.

The OLE Document Protocol

Keep in mind that OLE documents are but one feature of OLE. As such, the OLE document specification describes how OLE document containers and servers should behave. That is, it specifies what interfaces must be implemented on each side of the OLE document fence and the context in which each interface should be used.

There's nothing really special about the OLE document protocol. It's just a description of how to use a couple handfuls of COM interfaces to coordinate a higher-level protocol. MFC provides an implementation of OLE documents. Let's take a look at that now.

MFC's Support for OLE Documents

AppWizard includes several buttons for creating OLE document container and server apps. So just what do those buttons do? Here's where we find out.

MFC's support for OLE documents is heavily rooted in its document/view architecture. And why not? The document/view architecture specifies a consistent model for managing and rendering data. And what are the elements of a compound document but more (albeit different kinds of) data?

MFC enhances the document/view framework to implement OLE documents. The basic idea is that the MFC OLE document class manages a set of objects derived from `CDocItem`. On the container side, the document manages `COleClientItems`, and on the server side, the document manages `COleServerItems`. Let's examine the base classes briefly.

The Base Classes: `CDocItem` and `COleDocument`

`CDocItem` is the base class representing the items held in an MFC-based OLE document application. MFC's OLE document-based document class, `COleDocument`, manages a list of `CDocItems`. The `CDocItem` class is related to the document in the same way that a view is related to a document. That is, from the document, you can always iterate through the list of `CDocItem` objects. Also, given a `CDocItem`, you can always navigate back to the document.

Other than manage this connection with the document, `CDocItem` doesn't do a whole lot. `CDocItem` is completely devoid of any interfaces; that's left up to the derived classes `COleClientItem` and `COleServerItem`.

The `COleDocument` class doesn't implement any interfaces, either. Its job in life is to manage the list of `CDocItem` objects. Within the MFC hierarchy, `COleDocument`

derives from CDocument. Therefore COleDocument has all the regular support for the document/view architecture *and* it maintains a list of CDocItem objects.

COleDocument has several interesting member functions and features for navigating the list of document items and for dealing with Structured Storage.

COleDocument manages a single list of CDocItem objects, called `m_docItemList`. COleDocument has the following functions for managing that list: `GetStartPosition()`, `GetNextItem()`, `GetNextClientItem()`, `GetNextServerItem()`, `AddItem()`, and `RemoveItem()`.

`GetStartPosition()` and `GetNextItem()` are just type-safe wrappers for iterating through the list. `AddItem()` and `RemoveItem()` add and delete document items from the list using the regular MFC list management functions.

One interesting thing to note here is that COleDocument contains a single list that holds both COleClientItem objects and COleServerItem objects. If the application is both a container and a server, the application can mix these items within the list, creating a heterogeneous list. However, the application often needs to access all the items of a particular kind at once. For example, to perform rendering, a container's view needs to iterate through the COleClientItem objects, avoiding all the COleServerItem objects. That's where the specialized functions `GetNextClientItem()` and `GetNextServerItem()` come in. These functions let the application iterate through all the items of a particular kind. They work by calling an undocumented function in COleDocument: `GetNextItemOfKind()`. `GetNextItemOfKind()` extends `GetNextItem()` by taking a CRuntimeClass as a parameter. `GetNextItemOfKind()` iterates through the list returning the items that match the type represented by the CRuntimeClass.

Before we stop talking about the CDocItem list, notice that it holds any kind of object derived from CDocItem. That means you can derive your own classes from CDocItem and include them with COleClientItem and COleServerItem objects.

To help manage Structured Storage, COleDocument has a function called `EnableCompoundFile()`. There's not too much to this function. Calling `EnableCompoundFile()` turns on the document's Structured Storage by setting a Boolean flag in the COleDocument: `m_bCompoundFile`. The document checks this variable from time to time when it has to decide whether to use a regular file or an OLE Structured Storage object.

Finally, COleDocument maintains a root storage (that is, an IStorage pointer), into which it places native data as well as linked and embedded objects. The variable is called `m_lpRootStorage`. We'll see how it is used when we examine the interaction between OLE document containers and OLE document servers. Because the root storage is just an IStorage interface, you're free to create your own streams and storages off of it (in addition to the ones created by MFC as you add OLE items to your document).

OLE Document Containers the MFC Way

Believe it or not, MFC's OLE document container support is actually quite simple. Two classes lie at its heart: COleClientItem and COleDocument.

COleClientItem

Remember, COleDocument maintains a list of CDocItems. COleClientItems are derived from CDocItem, so COleDocument can include them in its list. You'll find most of the support for OLE document containers in COleClientItem. COleClientItem serves two purposes: (1) it manages the communication with the server application on the other end, and (2) it serves as an anchor for each embedded or linked item in your document.

In an MFC-based OLE document container, the document holds one COleClientItem for each embedded or linked item. For example, if an (MFC-based) OLE document contains an Excel spreadsheet, a Paintbrush painting, and a video clip, that OLE document is managing three COleClientItems.

COleClientItem meets the OLE document protocol standard by implementing the following interfaces: IOleClientSite, IAdviseSink, IOleWindow, and IOleInPlaceSite.

IOleClientSite

By exposing IOleClientSite, an OLE document container provides a channel so that the OLE document content object can communicate with the container. Here's the interface:

```
interface IOleClientSite : public IUnknown {
public:
    virtual HRESULT SaveObject( void ) = 0;
    virtual HRESULT GetMoniker( DWORD dwAssign,
                               DWORD dwWhichMoniker,
                               IMoniker FAR * FAR *ppmk ) = 0;
    virtual HRESULT GetContainer(IOleContainer FAR *FAR *ppContainer) = 0;
    virtual HRESULT ShowObject( void ) = 0;
    virtual HRESULT OnShowWindow(BOOL fShow) = 0;
    virtual HRESULT RequestNewObjectLayout( void ) = 0;
};
```

We list the interface here for reference. We'll look at these functions as we watch an MFC OLE document container and server do their stuff.

COleClientItem also implements IAdviseSink.

IAdviseSink

COleClientSite implements IAdviseSink so that OLE document content objects can notify OLE document containers about a change in the content object. You may recall

from the UDT discussion that `IDataObject::DAdvise()` takes an `IAdviseSink` as a parameter. Here's `IAdviseSink`:

```
interface IAdviseSink : public IUnknown {
public:
    virtual void OnDataChange(FORMATETC FAR *pFormatetc,
                           STGMEDIUM FAR *pStgmed) = 0;
    virtual void OnViewChange(DWORD dwAspect, LONG lindex) = 0;
    virtual void OnRename(IMoniker FAR *pmk) = 0;
    virtual void OnSave( void ) = 0;
    virtual void OnClose( void ) = 0;
};
```

Once this interface is plugged into the content object's `IDataObject` interface (using `IDataObject::DAdvise()`), the content object can notify the container about changes. For example, if the server's presentation data changes, all the content object has to do is call `IAdviseSink::OnDataChange()` to tell the client that something changed (at which point the container may choose to perform an action, such as redrawing its presentation).

IOleWindow

`IOleWindow` allows both OLE document containers and content objects to share the handles of the window participating in in-place activation. `IOleWindow` also aids in managing context-sensitive help between OLE compound-document applications. Here's the actual interface:

```
interface IOleWindow : public IUnknown {
    virtual HRESULT GetWindow(HWND FAR *phwnd) = 0;
    virtual HRESULT ContextSensitiveHelp(BOOL fEnterMode) = 0;
};
```

Several other in-place-activation interfaces are derived from the `IOleWindow` interface, including `IOleInPlaceSite`, the final interface implemented by `COleClientItem`.

IOleInPlaceSite

`COleClientItem` implements `IOleInPlaceSite` to coordinate the interactions between the container and the object's in-place client site. The functions in this interface manage in-place objects. For example, when the user tries to activate a content object, the content object uses `IOleInPlaceSite` to (1) determine if an object can be activated, (2) manage activation, and (3) manage deactivation. Content objects also use `IOleInPlaceSite` to notify the container *when* one of its objects is being activated and to inform the container to prepare for menu negotiation (that is, transferring the server's top-level menu to the container). In addition, `IOleInPlaceSite` includes methods

for providing the in-place object with the window object hierarchy and supplies the position (in the parent window) where the object should place its in-place activation window. Finally, IOleInPlaceSite manages how the container scrolls the object, manages its undo state, and notifies the object when its borders have changed. Again, this is a rather large interface; thank goodness it's already implemented by MFC. Listing 13-4 shows IOleInPlaceSite.

Listing 13-4. The IOleInPlaceSite interface

```
interface IOleInPlaceSite : public IOleWindow {
public:
    virtual HRESULT CanInPlaceActivate( void ) = 0;
    virtual HRESULT OnInPlaceActivate( void ) = 0;
    virtual HRESULT OnUIActivate( void ) = 0;
    virtual HRESULT GetWindowContext(IOleInPlaceFrame FAR *FAR *ppFrame,
                                    IOleInPlaceUIWindow FAR *FAR *ppDoc,
                                    LPRECT lprcPosRect,
                                    LPRECT lprcClipRect,
                                    LPOLEINPLACEFRAMEINFO lpFrameInfo) = 0;
    virtual HRESULT Scroll(SIZE scrollExtent) = 0;
    virtual HRESULT OnUIDeactivate(BOOL fUndoable) = 0;
    virtual HRESULT OnInPlaceDeactivate( void ) = 0;
    virtual HRESULT DiscardUndoState( void ) = 0;
    virtual HRESULT DeactivateAndUndo( void ) = 0;
    virtual HRESULT OnPosRectChange(LPCRECT lprcPosRect) = 0;
};
```

COleDocument and COleClientItem are the only classes you need to implement simple embedding and linking containers in MFC. A simple embedding container is one that contains only embeddings and simple links. However, from time to time some other application may want to create links to items embedded in your container or to items embedded in some other container.

For example, imagine that you have a word processor document that has a spreadsheet embedded in it. By supporting links to embedded objects, an application can paste a link to the spreadsheet (which is contained in the word processor's document). That way, the application can use the information contained in the spreadsheet without knowing where the word processor originally got it. That's a whole new story.

MFC provides another document class to support this feature: COleLinkingDoc.

COleLinkingDoc

The next class down MFC's OLE document hierarchy is COleLinkingDoc. COleLinkingDoc derives from COleDocument, so it manages a list of COleClientItems the same way as COleDocument. However, COleLinkingDoc implements several other COM interfaces

to enable linking to items that are embedded elsewhere. Those interfaces are **IPersistFile**, **IParseDisplayName**, **IOleContainer**, and **IOleItemContainer**.

IPersistFile

By implementing **IPersistFile**, an object tells its client that it can load and save itself in a disk file (rather than in a storage object). The object implementing **IPersistFile** is responsible for opening its own disk file, because the act of opening a file may vary from one application to another. **COleLinkingDocument** implements this interface to let other OLE documents link to MFC OLE document containers that already contain embeddings. That is, MFC OLE document containers that hold embeddings can themselves be linked to some other OLE document.

```
interface IPersistFile : public IPersist {
    virtual HRESULT IsDirty( void ) = 0;
    virtual HRESULT Load( LPOLESTR pszFileName,
                         DWORD dwMode ) = 0;
    virtual HRESULT Save(LPOLESTR pszFileName,
                         BOOL fRemember) = 0;
    virtual HRESULT SaveCompleted(LPOLESTR pszFileName) = 0;
    virtual HRESULT GetCurFile(LPOLESTR FAR *ppszFileName) = 0;
};
```

IPersistFile is usually implemented in objects that include other interfaces that support *monikers*.

Uh-oh—another new term. So what is a moniker? When OLE document content objects are linked, the actual data is stored in some other location. A moniker manages the information necessary for the OLE document container to locate the linked data and *bind* to it. As you can probably guess, a moniker object implements the **IMoniker** interface.

MFC OLE document containers deal with two kinds of monikers: file monikers and item monikers. File monikers identify data stored in a file. To that end, file monikers contain a path to the linked data. Item monikers represent a finer granularity: they identify data items within a file.

What does all this have to do with MFC and OLE documents? When an OLE document container creates a simple link to another file, the document uses a file moniker. That's no big deal as far as MFC support is concerned. However, when working with a linking container—one that can contain links to embedded objects somewhere else—MFC uses item monikers. A great example of how you might use an item moniker is to name a range of spreadsheet cells within a document. You may not want to link the whole spreadsheet, so instead you link a range of cells. That range is identified by an item moniker.

COleLinkingDoc implements three other interfaces for implementing an OLE linking document: **IParseDisplayName**, **IOleContainer**, and **IOleItemContainer**.

IParseDisplayName

IParseDisplayName parses a human-readable display name string so it may be converted into a moniker. The interface has only one function:

```
interface IParseDisplayName : public IUnknown {
    virtual HRESULT ParseDisplayName(IBindCtx FAR *pbc,
                                    LPOLESTR pszDisplayName,
                                    ULONG FAR *pchEaten,
                                    IMoniker FAR *FAR *ppmkOut) = 0;
};
```

IOleContainer

By implementing **IOleContainer**, applications that support links to embedded objects provide object enumeration, name parsing, and silent updates of link sources. Simple, nonlinking containers do not need to implement **IOleContainer**, so it's implemented in **COleLinkingDoc** (instead of **COleDocument**).

Here's the interface:

```
interface IOleContainer : public IParseDisplayName {
    virtual HRESULT EnumObjects(DWORD grfFlags,
                               IEnumUnknown FAR * FAR *ppenum) = 0;
    virtual HRESULT LockContainer(BOOL fLock) = 0;
};
```

IOleItemContainer

IOleItemContainer is used by item monikers once they are bound to their objects.

```
interface IOleItemContainer : public IOleContainer {
    virtual HRESULT GetObject(LPOLESTR pszItem,
                           DWORD dwSpeedNeeded,
                           IBindCtx FAR *pbc,
                           REFIID riid,
                           void FAR * FAR *ppvObject) = 0;
    virtual HRESULT GetObjectStorage(LPOLESTR pszItem,
                                   IBindCtx FAR *pbc,
                                   REFIID riid,
                                   void FAR * FAR *ppvStorage) = 0;
    virtual HRESULT IsRunning(LPOLESTR pszItem) = 0;
};
```

Moniker providers (for an example, an OLE document server that supports linking) must implement `IOleItemContainer`. A moniker provider supplies monikers that identify objects and makes them accessible to moniker clients (that is, OLE document containers).

OLE document servers that support linking define their own naming scheme for identifying the objects within a container. The naming scheme is left to the discretion of the server. For example, embedded objects in a document could be identified with “Embedding1,” “Embedding2,” and so forth (in fact, this is how MFC containers name embeddings). On the other hand, spreadsheet cell ranges might be identified by names like “A1:B4”. (Ranges of cells in a spreadsheet are examples of *pseudo-objects*. They do not have their own persistent storage; they simply represent a portion of the container’s internal state.) An application creates an item moniker that represents an object’s name using the `CreateItemMoniker()` API function and hands it out to a moniker client. When an item moniker is bound, the container’s implementation of `IOleItemContainer` must be able to take a name and retrieve the corresponding object.

That does it for MFC’s support for OLE document containers. Let’s look now at MFC’s OLE document server support.

OLE Document Servers the MFC Way

MFC’s OLE document server architecture is very similar in structure to MFC’s container support. An MFC OLE document server derives its document from `COleServerDoc` (instead of `COleDocument` or `COleLinkingDoc`), and the document maintains `COleServerItem` objects in its `CDocItem` list.

COleServerDoc

`COleServerDoc` is another step down the `CDocument` hierarchy, so it has all the functionality necessary for maintaining a list of `CDocItems` (as well as the linking-container support provided by `COleLinkingDoc`). However, `COleServerDoc` can support `COleServerItems` in the list as well.

At the minimum, an embedded OLE document content object has to support `IOleObject`, `IPersistStorage`, and `IDataObject`. If you look at `COleServerDoc`’s interface map, you’ll see that `COleServerDoc` implements these interfaces. `COleServerDoc` also supports in-place activation by supporting `IOleInPlaceObject` and `IOleInPlaceActiveObject`.

IPersistStorage

COleServerDoc implements IPersistStorage to tell OLE document containers that it supports persistence to an OLE Structured Storage. Take a quick look at IPersistStorage and notice the nature of its member functions:

```
interface IPersistStorage : public IPersist {
public:
    virtual HRESULT IsDirty( void ) = 0;
    virtual HRESULT InitNew(IStorage FAR *pStg) = 0;
    virtual HRESULT Load(IStorage FAR *pStg) = 0;
    virtual HRESULT Save(IStorage FAR *pStgSave,
                         BOOL fSameAsLoad) = 0;
    virtual HRESULT SaveCompleted(IStorage FAR *pStgNew) = 0;
    virtual HRESULT HandsOffStorage(void) = 0;
};
```

OLE document containers acquire the content object's IPersistStorage and use the interface to store the OLE document content objects.

IOleObject

The second mandatory interface for OLE document content objects is IOleObject. IOleObject handles the bulk of the communication responsibilities for the content object. Take a look at the interface, shown in Listing 13-5.

Listing 13-5. The IOleObject interface

```
interface IOleObject : public IUnknown {
public:
    virtual HRESULT SetClientSite(IOleClientSite FAR *pClientSite) = 0;
    virtual HRESULT GetClientSite(IOleClientSite FAR *FAR
                                 *ppClientSite) = 0;
    virtual HRESULT SetHostNames(LPCOLESTR szContainerApp,
                                LPCOLESTR szContainerObj) = 0;
    virtual HRESULT Close(DWORD dwSaveOption) = 0;
    virtual HRESULT SetMoniker(DWORD dwWhichMoniker,
                             IMoniker FAR *pmk) = 0;
    virtual HRESULT GetMoniker(DWORD dwAssign,
                             DWORD dwWhichMoniker,
                             IMoniker FAR *FAR *ppmk) = 0;
    virtual HRESULT InitFromData(IDataObject FAR *pDataObject,
                               BOOL fCreation,
                               DWORD dwReserved) = 0;
    virtual HRESULT GetClipboardData(DWORD dwReserved,
                                 IDataObject FAR *FAR
                                 *ppDataObject) = 0;
```

```
virtual HRESULT DoVerb(LONG iVerb,
                      LPMMSG lpmsg,
                      IOleClientSite FAR *pActiveSite,
                      LONG lindex,
                      HWND hwndParent,
                      LPCRECT lprcPosRect) = 0;

virtual HRESULT EnumVerbs(IEnumOLEVERB FAR *FAR *ppEnumOleVerb) = 0;
virtual HRESULT Update( void ) = 0;
virtual HRESULT IsUpToDate( void ) = 0;
virtual HRESULT GetUserClassID(CLSID FAR *pClsid) = 0;
virtual HRESULT GetUserType(DWORD dwFormOfType,
                           LPOLESTR FAR *pszUserType) = 0;
virtual HRESULT SetExtent(DWORD dwDrawAspect,
                         SIZEL FAR *psizel) = 0;
virtual HRESULT GetExtent(DWORD dwDrawAspect,
                         SIZEL FAR *psizel) = 0;
virtual HRESULT Advise(IAdviseSink FAR *pAdvSink,
                      DWORD FAR *pdwConnection) = 0;
virtual HRESULT Unadvise(DWORD dwConnection) = 0;
virtual HRESULT EnumAdvise(IEnumSTATDATA FAR *FAR *ppenumAdvise) = 0;
virtual HRESULT GetMiscStatus(DWORD dwAspect,
                             DWORD FAR *pdwStatus) = 0;
virtual HRESULT SetColorScheme(LOGPALETTE FAR *pLogpal) = 0;
};
```

Here's another big interface. Again, it's good that MFC implements this interface. IOleObject clocks in at 21 functions (not including the IUnknown functions). MFC actually splits this interface between the COleServerItem class and the COleServerDoc class. We'll see how MFC implements this interface when we examine the interactions between an MFC OLE document container and server.

IDataObject

You had a chance to see `IDataObject` in Chapter 12. Though you can use `IDataObject` for plain-vanilla Clipboard and drag-and-drop operations, `IDataObject`'s natural habitat is within an OLE document server. Listing 13-6 shows the interface.

Listing 13-6. The IDataObject interface

```

virtual    HRESULT   SetData(FORMATETC *pformatetc,
                           STGMEDIUM* pmedium,
                           BOOL fRelease) = 0;
virtual    HRESULT   EnumFormatEtc(DWORD dwDirection,
                                   FORMATETC* pformatetc) = 0;
virtual    HRESULT   DAdvise(FORMATETC* pformatetc,
                           DWORD grfAdvf,
                           IAdviseSink *pAdvSink,
                           DWORD pdwConnection) = 0;
virtual    HRESULT   DUnadvise(DWORD dwConnection) = 0;
virtual    HRESULT   EnumDAdvise(IEnumSTATDATA *ppenumAdvise) = 0;
}

```

In addition to the three mandatory interfaces, MFC's OLE document server classes include support for in-place activation: the **IOleInPlaceObject** and **IOleInPlaceActiveObject** interfaces.

IOleInPlaceObject

IOleInPlaceObject gives the container a way to activate and deactivate in-place objects, as well as manage how much of the in-place object should be visible. Here's **IOleInPlaceObject**. Notice that it derives from **IOleWindow**:

```

interface IOleInPlaceObject : public IOleWindow
    virtual HRESULT InPlaceDeactivate( void ) = 0;
    virtual HRESULT UIActivate( void ) = 0;
    virtual HRESULT SetObjectRects(LPCRECT lprcPosRect,
                                  LPCRECT lprcClipRect) = 0;
    virtual HRESULT ReactivateAndUndo( void ) = 0;
};


```

The last interface implemented by **COleServerDoc** is **IOleInPlaceActiveObject**.

IOleInPlaceActiveObject

OLE document content objects implement **IOleInPlaceActiveObject** to provide a direct channel between (1) an in-place object, (2) the outermost frame window of the object's server application, and (3) the document window within the container application. The interface handles translating messages, managing the state of the frame window (activated or deactivated), and managing the state of the document window (activated or deactivated). The interface also informs the content object when it needs to resize its borders, and it manages modeless dialog boxes. Here it is:

```

interface IOleInPlaceActiveObject : public IOleWindow {
    virtual HRESULT TranslateAccelerator(LPMSG lpmsg) = 0;
};


```

```

virtual HRESULT OnFrameWindowActivate(BOOL fActivate) = 0;
virtual HRESULT OnDocWindowActivate(BOOL fActivate) = 0;
virtual HRESULT ResizeBorder(LPCRECT prcBorder,
                           IOleInPlaceUIWindow FAR *pUIWindow,
                           BOOL fFrameWindow) = 0;
virtual HRESULT EnableModeless(BOOL fEnable) = 0;
};

}

```

Notice that `IOleInPlaceActiveObject` also derives from `IOleWindow`, so it has the `GetWindow()` and `ContextSensitiveHelp()` functions.

COleServerItem

`COleServerItem` represents the other half of MFC's support for OLE document servers. `COleServerItem` implements only two interfaces: `IOleObject` and `IDataObject`. MFC splits the implementation of these interfaces between `COleServerDoc` and `COleServerItem`. We could say more about `COleServerItem` here, however, it is more interesting to see the classes in action. Let's watch as an MFC OLE document container and server do their stuff.

The Container/Server Dance (Embedding)

Spelling out each individual piece of MFC's OLE document implementation would get very dull before long. Instead, it's much more interesting to watch the interaction between an OLE document container and server when they're running. Let's track how things work between an embedded OLE document container and server. In the process we'll see the following:

- How the container document creates content objects.
- How the container and server deal with Structured Storage.
- How the container and server communicate.

The Container: Creating a New File

For the container, the action begins as soon as the application tries to create a new file. This can happen when (1) the application starts or (2) the user selects New from the File menu. MFC responds by calling `CWinApp`'s `OnFileNew()` function. Let's imagine that the user has selected FileNew. The framework calls `CWinApp::OnFileNew()`

in response to the File|New menu item. CWinApp::OnFileNew() performs several steps to get the new document up and running. The call chain goes like this:

- CWinApp::OnFileNew() calls the document manager's OnFileNew() function.
- The document manager's OnFileNew() handler gets a document template and opens the document file by calling the template's OpenDocumentFile().
- The document template's OpenDocumentFile() calls the document template's CreateNewDocument() function.
- This culminates in a call to your document's OnNewDocument() function.

So far, this is just regular MFC document/view stuff, which was covered in Chapters 7 and 8.

If you let AppWizard generate your OLE document server program, AppWizard inserts a call to EnableCompoundFile() in your document's constructor. This sets the COleDocument's *m_bEmbedded* flag to TRUE.

AppWizard also inserts a call to COleDocument::OnNewDocument() in your document's OnNewDocument() member. If compound files have been enabled for the document, COleDocument::OnNewDocument() releases the existing *m_lpRootStorage* and creates a new one using the OLE API function StgCreateDocFile(). After Structured Storage is up and running, CMultiDocTemplate::OpenDocumentFile() calls InitialUpdateFrame() to update all the views.

Once the container's Structured Storage is open, it's ready to accept new OLE document items.

Adding Container Items

MFC's OLE document implementation involves managing a list of COleClientItems. But where do these come from? They usually come from one of three places: (1) the user selects some variation of "Insert New Object" from the menu, (2) the user selects Paste or Paste Special from the Edit menu, or (3) the user selects Paste Link from the Edit menu. Each of these choices involves constructing a COleClientItem. Here's how MFC goes about doing it.

Constructing COleClientItems

Before you can add a COleClientItem to the document, you've got to construct one. COleClientItem's constructor takes a single parameter—a pointer to a COleDocument:

```
COleClientItem::COleClientItem(COleDocument* pContainerDoc);
```

COleClientItem's constructor initializes member variables to their default values, which is usually NULL, void, or FALSE. Then the constructor adds the COleClientItem to the document using the COleDocument's AddItem() function.

That was simple enough. But what can you do with a COleClientItem once it's been constructed? Let's examine how to create a new COleClientItem from scratch using MFC's COleInsertDlg class.

The Container: Inserting a New Item

Most containers have a menu option for inserting a new OLE document object. For example, Word for Windows includes an "InsertObject" menu. Using AppWizard to generate your application will yield an "Insert New Object" option under the Edit menu. The easiest way to handle the Insert New Object menu option is by invoking the COleInsertDlg common dialog and using it to create an OLE document content object.

From a developer's perspective, using the COleInsertDlg is very easy. Just declare an instance of it and call DoModal() to invoke it. COleInsertDlg is really a wrapper for the OLE API function ::OleUIInsertObject(). ::OleUIInsertObject() invokes the standard Insert Object dialog box. That way, users can select an object source and class name, as well as the option of displaying the object's full content or just its icon.

Figure 13-4 shows the Insert Object dialog box represented by COleInsertDlg.

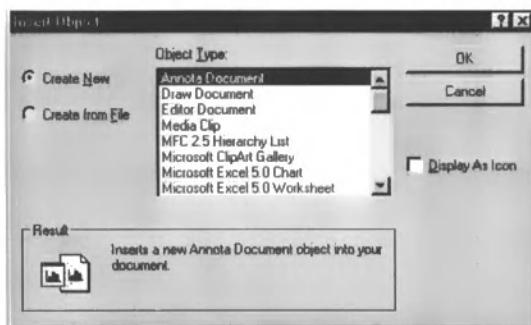


Figure 13-4. The COleInsertDlg

After successfully calling COleInsertDlg::DoModal(), you can use CreateItem(), COleInsertDlg's convenient function for creating client items:

```
BOOL CreateItem(COleClientItem* pItem);
```

Just construct a COleClientItem (which connects it to a COleDocument). Then feed your newly constructed COleClientItem to COleInsertDlg's CreateItem() function.

CreateItem() handles all the details of creating a new substorage for the COleClientItem, firing up the server and establishing the COM connections necessary to satisfy the OLE document protocol.

Listing 13-7 shows the code that does the trick. In fact, this is just what the AppWizard-generated code does.

Listing 13-7. Code for inserting an OLE document content object into a container

```
void OnEditInsertNewObject() {
    COleInsertDialog dlg;
    if (dlg.DoModal() != IDOK)
        return;

    CContainerItem* pItem = NULL;
    TRY
    {
        // Create new item connected to this document.
        CContainerDoc* pDoc = GetDocument();
        pItem = new CContainerItem(pDoc);

        // Initialize the item from the dialog data.
        if (!dlg.CreateItem(pItem))
            AfxThrowMemoryException(); // any exception will do
        ASSERT_VALID(pItem);
    }

    CATCH(CException, e) {
        if (pItem != NULL) {
            pItem->Delete();
        }
        AfxMessageBox("Couldn't create OLE Document content object");
    }
    END_CATCH

    EndWaitCursor();
};

}
```

Is that all it takes to create a content object in MFC? Wow! Obviously, MFC's doing a lot of work. Let's take a look inside COleInsertDlg::CreateItem().

Inside CreateItem()

If the new client item is a simple embedding, COleInsertDlg::CreateItem() simply delegates the work to COleClientItem::CreateNewItem(). Other options include creating an item from a file (COleClientItem::CreateFromFile()) or linking an item from a file (COleClientItem::CreateLinkFromFile()).

One of the primary responsibilities of an OLE document container is managing and organizing the various content objects inside a file. The first thing COleClientItem::CreateNewItem() does is create a substorage for the content object. Each substorage has to have a unique name, so CreateNewItem() has to come up with that name. COleDocument maintains a count of the number of items contained in the document. Each time a new client item is added to the document, the document increments this counter. The new client item is branded with the next number. The client item uses that number to uniquely specify the client item's substorage. Each new client item is stored using the name "EmbeddingX", where X is the number of the item.

Next, COleClientItem::CreateNewItem() creates a new storage for the item by calling COleClientItem::GetItemStorage(). GetItemStorage() examines the document's m_bCompoundFile flag (remember, that was set by calling COleDocument::EnableCompoundFile()). If the COleDocument uses a compound file, GetItemStorage() calls GetItemStorageCompound(). Otherwise, GetItemStorage() uses GetItemStorageFlat() to create the storage.

GetItemStorageCompound() uses the document's m_lpStorage to create a new storage. Remember, m_lpStorage is a pointer to an IStorage interface. IStorage has a member function—CreateStorage()—that creates a new substorage. Then GetItemStorageCompound() sets the COleClientItem's m_lpStorage member to the newly created storage. This is the IStorage that the content object will use to store its data.

If the application didn't enable compound files, then the client item uses GetItemStorageFlat() to create a new storage. First, GetItemStorageFlat() uses the OLE API function CreateILockBytesOnHGlobal() to create an object that implements ILockBytes in memory. Then GetItemStorageFlat() feeds the ILockBytes pointer to the OLE API function StgCreateDocfileOnILockBytes(), which hands back an IStorage pointer, which holds the content object's data.

Either way, the COleClientItem has an IStorage pointer to work with. So what does CreateNewItem() do with this storage? CreateNewItem() uses it in a call to OleCreate():

```
WINOLEAPI OleCreate(REFCLSID rclsid, REFIID riid, DWORD renderopt,
                    LPFORMATETC pFormatEtc, LPOLECLIENTSITE pClientSite,
                    LPSTORAGE pStg, LPVOID FAR* ppvObj);
```

OleCreate() is basically an enhanced version of CoCreateInstance(). OleCreate() takes seven parameters:

1. The GUID representing the desired COM class.
2. The GUID representing the requested interface.

3. The render option describing how the object will be cached.
4. A pointer to a FORMATETC structure, which helps describe how the object will be cached.
5. A pointer to a client site (an IOleClientSite interface).
6. A pointer to a storage (an IStorage interface).
7. A place to put the requested interface pointer.

Here's how COleInsertDlg::CreateItem() calls OleCreate():

```
OleCreate(clsid, IID_IUnknown, render,
          lpFormatEtc, lpClientSite, m_lpStorage,
          (LPLP)&m_lpObject);
```

The class ID was passed to CreateNewItem() by COleInsertDlg::CreateItem(). The class ID is from the program ID chosen by the user (the selection that was highlighted in the list box). Also notice that CreateNewItem() asks for IUnknown. To specify how to render the cache, CreateNewItem() defaults to OLERENDER_DRAW, which indicates that a drawing will be rendered on the screen. The FORMATETC pointer is NULL (the default from CreateNewItem()'s parameter list). CreateNewItem() uses its GetInterface() function to get the IOleClientSite interface from the COleClientItem object. The IStorage pointer is the newly created substORAGE off the container document's root storage.

OleCreate() performs the following steps:

1. OleCreate() creates the COM object using CoCreateInstance().
2. OleCreate() calls QueryInterface() to get the content object's IPersistStorage interface.
3. OleCreate() calls IPersistStorage::InitNew() to pass the content object an IStorage pointer. The content object uses that substORAGE to store its data in the container's document.
4. OleCreate() then passes an IOleClientSite interface pointer to the content object and initializes the cache.

If you follow through both sides of the equation, you'll be able to see how the container's call to OleCreate() starts up the server application, acquires the class factory for the content object, and creates the content object. If the server is an MFC application, then the document is usually constructed.

Once the document is constructed, the container tries to acquire the content object's IPersistStorage interface and calls IPersistStorage::InitNew(), passing the

newly created substorage. If the server application is an MFC application, then the call ends up in COleServerDoc::XPersistStorage::InitNew().

On the server side, the COleServerDoc handles IPersistStorage::InitNew() by calling COleServerDoc::OnNewEmbedding(). COleServerDoc::OnNewEmbedding() cleans out the document by deleting its contents. This includes deleting any previous links by calling COleLinkingDoc::DeleteContents(). In turn, COleLinkingDoc::DeleteContents() calls COleDocument::DeleteContents() to delete any COleClientItem objects. Then the document deletes any COleServerItem objects. Finally, COleServerDoc caches the client's IStorage interface in a member variable: m_lpRootStorage. COleServerDoc needs that root storage so it can access the container's storage (to read and write data). Then OnNewEmbedding() calls OnNewDocument() to initialize a new document.

At this point, the server does whatever is necessary to initialize the specific document. For example, the Scribble tutorial that comes with MFC deletes all the strokes from the document.

Before finishing, OnNewEmbedding() sets the document-is-modified flag and calls SendInitialUpdate() to update all the views. OnNewEmbedding() returns to COleServerDoc::XPersistStorage::InitNew(), which returns control to the container (which is still in the throes of OleCreate()).

Next, OleCreate() calls the content object's IOleObject::SetClientSite() so the content object has a chance to save the container's client site. If the server is an MFC-based application, this ends up in COleServerDoc::XOleObject::SetClientSite(). The COleServerDoc caches the client site in its m_lpClientSite variable. Then control volleys over to the container again, which is finishing the calls to OleCreate().

Back in the container (after the call to OleCreate() is done), the COleClientItem has to finish connecting to the content object. COleClientItem() has a function named FinishCreate() that does just that. FinishCreate() takes the IUnknown pointer returned by OleCreate() and calls QueryInterface() to get the content object's IOleObject interface. FinishCreate() saves the IOleObject interface in a member variable within COleClientItem. The client item uses this interface frequently during the course of its lifetime. Then FinishCreate() retrieves the content object's IVViewObject2 interface. Finally, FinishCreate() plugs the client item's IAdviseSink into the content object's IDataObject (using IDataObject::DAdvise()) and into the content object's IVViewObject2 interface (using IVViewObject2::SetAdvise()). Now the container will be able to talk intelligently to the content object, and the content object will be able to advise the container of any changes.

Once the container has created a COleClientItem, the container can start working with it. The first thing a container usually does is to activate the object by calling DoVerb() in the content object's IOleObject interface using the constant OLEIVERB_PRIMARY. This process is rather involved, but it's very interesting to watch!

DoVerb() and In-Place Activation

In-place activation occurs whenever a container calls the content object's IOleObject::DoVerb() function. This is interesting to watch because it is where all the menu hocus-pocus happens.

By this point, the container and the content object are engaged in a rich conversation, and the COleClientItem has already cached the content object's IOleObject interface. Remember, IOleObject is the main interface the client uses to talk with the content object. You'll find DoVerb() inside IOleObject.

To execute a verb on the server side, COleClientItem has a function called DoVerb(). COleClientItem::DoVerb() calls COleClientItem::Activate(). Activate takes three parameters: the number representing the verb, a pointer to a CView, and a pointer to the message that invoked the verb.

IOleObject::DoVerb() takes several parameters, which COleClientItem::Activate() has to package up:

```
HRESULT DoVerb(LONG lVerb, // verb to execute
               LPMSG lpmsg, // message that invoked the verb
               IOleClientSite FAR *pActiveSite, // container's client site
               LONG lindex, // reserved
               HWND hwndParent, // parent window
               LPCRECT lprcPosRect); // position of inplace item
```

Activate() first gets the item's position on the screen by calling COleClient::OnGetPosition(). (That's one of the functions the "CONTAIN" tutorial tells you to override. At least now you know when it's used.)

Activate() then gets the COleClientItem's IOleClientSite and the view's HWND. The COleClientItem holds a pointer to the content object's IOleObject interface, so the COleClientItem can make the direct interface call:

```
m_lpObject->DoVerb(nVerb,
                     lpClientSite, -1,
                     hWnd, lpPosRect);
```

Note: If the server's not already loaded, the default handler loads the server and calls the content object's IPersistStorage::Load() function. In an MFC-based OLE document server, this ends up in COleServerDoc::XPersistStorage::Load().

Cut to the content object. The COleClientItem's call to IOleObject::DoVerb() ends up in COleServerDoc::XOleObject::DoVerb(). Take a quick look at COleServerDoc::XOleObject::DoVerb()'s implementation; it ignores all the parameters except the verb:

```
ColeServerDoc::XOleObject::DoVerb(LONG iVerb,
                                  LPMMSG /*lpmsg*/,
                                  LPOLECLIENTSITE /*pActiveSite*/,
                                  LONG /*lindex*/,
                                  HWND /*hwndParent*/,
                                  LPCRECT /*lpPosRect*/)
```

`COleServerDoc::XOleObject::DoVerb()` retrieves the document's `COleServerItem` by calling `COleServerDoc::GetEmbeddedItem()` and delegates the work through the embedded item's `OnDoVerb()` member.

`COleServerItem::OnDoVerb()` is basically a switch statement that switches on the verb. `OnDoVerb()` handles the standard verbs Open, Show, and Hide. If the requested verb isn't one of the standard verbs and it's negative, `OnDoVerb()` returns `E_NOTIMPL`. If the requested verb isn't one of the standard verbs and it's positive, then `OnDoVerb()` defaults to executing the primary verb.

`COleServerItem::OnDoVerb()` is virtual, so if you feel like adding a custom verb to your OLE document server, this is the place to do it. Just override `COleServerItem::OnDoVerb()` and handle your custom verb, letting the framework handle the verb if it's not your custom verb.

We're examining what happens when the container tries to activate the item, so the verb is `OLEIVERB_PRIMARY`. This causes `OnDoVerb()` to call `COleServerItem::OnShow()`. `COleServerItem::OnShow()` turns around, retrieves the document, and calls `COleServerItem::ActivateInPlace()`. Wow—these function calls are going all over the place!

`COleServerItem::ActivateInPlace()` activates the item for in-place editing. This function performs all operations necessary for in-place activation, including (1) creating an in-place frame window, (2) positioning the in-place window, (3) hooking the container and server windows together so that menu commands are handled appropriately, (4) setting up shared menu controls, and (5) scrolling the item into view and setting the focus to the in-place frame window. Let's see how MFC performs each of these functions.

Creating the In-Place Frame

`COleServerItem::ActivateInPlace()` starts out by building the title out of the app name and the file type. For example, Scribble's embedded title is "Scribble—SCRIB".

Next, `ActivateInPlace()` tries to get the container's `IOleInPlaceSite` interface by calling `QueryInterface()` on the container's `IOleClientSite`. Once the container's `IOleInPlaceSite` interface is available, the content object uses `IOleInPlaceSite::CanInPlaceActivate()` to ask the container if it can "in-place activate."

On the client side, this ends up in `COleClientItem::XOleIPSite::OnInPlaceActivate()`, which calls `COleClientItem::OnActivate()`. `OnActivate()` brings the content object

out of the loaded state and into the running state by calling `OleLockRunning()`. Before leaving, `OnActivate()` calls the client item's `OnChange()` function (so the container can perform such operations as updating the screen).

Control then returns to the `COleServerDoc` (still in the middle of `ActivateInPlace()`). Next, `ActivateInPlace()` makes a call to the `IOleInPlaceSite`'s `GetWindow()` function.

On the container side, control ends up in `COleClientItem::XOleIPSite::GetWindow()`. The container simply returns the view's window handle to the content object.

Back on the server side, `COleServerDoc::ActivateInPlace()` creates a `CWnd` object out of the container's view. `ActivateInPlace()` uses the `CWnd` object to create an in-place frame (so that the container's view becomes the in-place frame's parent).

`COleServerDoc` has a member function—`CreateInPlaceFrame()`—for creating the in-place frame. `COleServerDoc::CreateInPlaceFrame()` retrieves the document template (remember, we're still inside the document object—we can get the document template). Fortunately, `CDocTemplate` has a function called `CreateOleFrame()`, which creates a `COleIPFrameWnd` and loads the appropriate server resources. `CreateInPlaceFrame()` uses the first view created for the document. This view is temporarily detached from the original frame and attached to the newly created in-place frame. Note that the functions that manage this are virtual by design: `CreateInPlaceFrame()` and `DestroyInPlaceFrame()`. These functions can be overridden if this default behavior of representing the view is not valid for your application. This is necessary if you have nonstandard or even semistandard arrangements of views. For example, if you wanted to use a splitter in your in-place session you would need to provide a new implementation of these functions.

Next, `ActivateInPlace()` calls the client's `IOleInPlaceSite::OnUIActivate()`. Whoops—back to the client.

Positioning the In-Place Window and Connecting the Server and Container Windows

Control moves over to `COleClientItem::XOleIPSite::OnUIActivate()` now. This function calls `COleClientItem::OnActivateUI()`. `OnActivateUI()` doesn't do much—it just makes sure the container view's `WS_CLIPCHILDREN` bit is set. This is necessary because the content in the object's in-place window should remain confined to the in-place window. Then `COleClientItem::XOleInPlaceSite::OnUIActivate()` retrieves the server's `IOleInPlaceObject` interface and uses it to get the server's in-place frame's `HWND` and saves it in `COleClientItem::m_hWndServer`.

Back in the server, the content object calls the container's `IOleInPlaceSite::GetWindowContext()` function. The content object uses `GetWindowContext()` for two kinds of information: (1) the position in the parent window where the object's in-place activation window should be placed and (2) the

window interfaces that form the window object hierarchy (those interfaces are the `OleInPlaceFrame` and `OleInPlaceUIWindow` interfaces). OLE uses the terms *frame window* and *document window* to describe the hierarchy. The frame window is the outside window (the one with menus), and the document window renders the presentation.

Control now lands in `COleClientItem::XOleIPSite::GetWindowContext()`. First, the client item sets up a position rectangle and a clipping rectangle so the content object has some idea about where to position its in-place window. `COleClientItem::XOleIPSite::GetWindowContext()` uses the virtual functions `COleClientItem::OnGetPosition()` and `COleClientItem::OnGetClipRect()` to determine these positions.

In addition to positioning information, the client item must give the content object a way to hook its in-place window into the container's windows. The content object requires an information structure of type `OLEINPLACEFRAMEINFO` and expects to see those two interfaces from the client: `IOleInPlaceFrame` and `IOleInPlaceUIWindow`. Let's see how they are set up.

Another `COleClientItem` virtual member function comes into play here: `OnGetWindowContext()`. This function (1) provides accelerator and window handle information to the container and (2) sets up the window hierarchy so that `COleClientItem::XOleIPSite::GetWindowContext()` can return `IOleInPlaceFrame` and `IOleInPlaceUIWindow` pointers that are hooked up to the correct windows.

To provide the extra information about the frame window, OLE defines a structure called `OLEINPLACEFRAMEINFO` for this:

```
typedef struct tagOIFI
{
    UINT cb;
    BOOL fMDIApp;
    HWND hwndFrame;
    HACCEL haccel;
    UINT cAccelEntries;
} OLEINPLACEFRAMEINFO, *LPOLEINPLACEFRAMEINFO;
```

Inside `COleClientItem::OnGetWindowContext()`, MFC uses the document template to retrieve the accelerator table for the information structure. The `hwndFrame` member is set to the container's main frame window handle.

Next, `OnGetWindowContext()` sets up the window hierarchy. If the container is an MDI application, `OnGetWindowContext()` sets up a pointer to the top-level frame (the `CMDIFrameWnd` object) as the main frame and a pointer to the active `CMDIChildWnd` object as the document window. If the container is an SDI application, `OnGetWindowContext()` sets up a pointer to the top-level frame (the `CFrameWnd` object) as the main frame window and assigns the value `NULL` to the document window pointer.

After calling `OnGetWindowContext()`, the `COleClientItem` has to provide the `IOleInPlaceFrame` and `IOleInPlaceUIWindow` interfaces. MFC has an undocumented helper class called `COleFrameHook` that implements and wraps `IOleInPlaceFrame` and `IOleInPlaceUIWindow`. After setting up the `OLEINPLACEFRAMEINFO` structure, MFC has to hook the `IOleInPlaceFrame` interface to the container's frame window and `IOleInPlaceUIWindow` to the container's MDI child frame (if this is an MDI application).

Back in `COleClientItem::OnGetWindowContext()`, the framework has set up pointers to the container's main frame and MDI child window (in the case of an MDI app)—now it's time to use those pointers. To implement `IOleInPlaceFrame` and `IOleInPlaceUIWindow`, `COleClientItem::XOleIPSite::GetWindowContext()` constructs two instances of `COleFrameHook`: one based on the outside frame and one based on the MDI child window (if the application is an MDI app). `COleFrameHook`'s constructor takes two parameters: a pointer to a `CFrameWnd` and a pointer to a `COleClientItem`. `COleFrameHook` has some important members: a pointer to a `COleClientItem` (`m_pActiveItem`), a pointer to a frame window (`m_pFrameWnd`), and a window handle (`m_hwnd`).

When constructing the outside frame hook, `COleFrameHook`'s constructor sets `m_pActiveItem` to the active `COleClientItem`, `m_pFrameWnd` to the outside frame, and `m_hwnd` to the outside frame's window handle. Buried inside MFC's `CFrameWnd` class, there's a `COleFrameHook` member called `m_pNotifyHook`. When the framework constructs the outside frame, it sets `m_pNotifyHook` to the `COleFrameHook` being constructed.

When constructing the MDI child frame hook, `COleFrameHook`'s constructor sets `m_pActiveItem` to the active `COleClientItem`, `m_pFrameWnd` to the MDI child frame, and `m_hwnd` to the MDI child frame's window handle. The constructor also sets `m_pNotifyHook` to the `COleFrameHook` being constructed.

Remember, the container site and the content object are talking via OLE interfaces. The content object expects to see two interface pointers, not two MFC objects. That is, the content object expects to see the outside frame-based `COleFrameHook` object through the `IOleInPlaceFrame` interface and the MDI frame-based `COleFrameHook` object through the `IOleInPlaceUIWindow` interface. `COleClientItem::XOleIPSite::GetWindowContext()` retrieves the appropriate interfaces for each of the newly constructed `COleFrameHook` objects using `GetInterface()` (remember, that handy function is available to all `CCmdTarget`-based classes that implement interface maps). Having retrieved the pointers, the client site can return the (correctly wired) interface pointers to the content object.

As you can see, there's a lot involved in setting up the window contexts between the container and the content object.

Control is now back in the hands of the content object. To keep the context in mind, remember that we're still executing the verb. The call stack at this point is this:

```
COleServerItem::OnDoVerb()
    which begat COleServerItem::OnShow()
        which begat COleServerDoc::ActivateInPlace()
```

Once the windows are hooked together, the content object needs to piece together the shared menu. Recall that `ActivateInPlace()` created a `COleIPFrameWnd` earlier. `ActivateInPlace()` calls the in-place frame window's `BuildSharedMenu()` function. Here's where the menu negotiating takes place. `BuildSharedMenu()` gets the active document's document template, from which `BuildSharedMenu()` gets the content object's in-place menu resources. `BuildSharedMenu()` then creates a menu handle by calling the Windows API function `::CreateMenu()`.

Remember the `COleFrameHook` class? We finally get to see how it's used! The content object has a pointer to the container's `IOleInPlaceUIWindow` interface. `IOleInPlaceUIWindow` has a member function for building the shared menus: it's called `InsertMenus()`. (Remember, `IOleInPlaceUIWindow` is implemented by `COleFrameHook`.)

Setting Up Menus and Toolbars

The content object calls `IOleInPlaceUIWindow::InsertMenus()`, which ends up in `COleFrameHook::XOleInPlaceFrame::InsertMenus()` within the container. The actual menu negotiation happens within the `COleClient` item. However, we're looking in a `COleFrameHook` object. How do we get a `COleClientItem` object? Remember that one of `COleFrameHook`'s members is the active `COleClientItem`; it was set when the `COleFrameHook` object was constructed. `COleFrameHook::XOleInPlaceFrame::InsertMenus()` uses the active `COleClientItem` and calls `COleClientItem::InsertMenus()` to insert the container application's menus into an empty menu.

MFC's implementation of `InsertMenus()` inserts the in-place container menus. That is, the function inserts the File, Container, and Window menu groups. This menu resource was attached to the container's document template using `CDocTemplate::SetContainerInfo()`.

Once the shared menu skeleton is constructed, `BuildSharedMenu()` calls `AfxMergeMenus()` to merge the container and server menus.

After the container is done with the menu, the server gets a chance to insert its item. Back in the server, `COleIPFrameWnd::BuildSharedMenu()` uses `AfxMergeMenus()` to add the server's menus to the shared menu.

Before leaving, COleIPFrameWnd::BuildSharedMenu() calls the OLE API function OleCreateMenuDescriptor() to create an OLE menu descriptor, an OLE-provided data structure that describes the menus. OLE needs this information when dispatching menu messages and commands. COleIPFrameWnd stores this structure in its m_hOleMenu member.

At this point, the content object installs its toolbar in the container and calls OnSetItemRects() to position the in-place editing frame window within the container application's frame window.

The content object then calls the container's IOleInPlaceFrame::SetActiveObject(), passing its IOleInPlaceActiveObject interface to the client. The content object's IOleInPlaceActiveObject interface provides a direct channel of communication between (1) the content object, (2) the server application's outermost frame window, and (3) the container's document window. This interface handles such user-interface issues as translating messages, the state of the frame window (activated or deactivated), and the state of the document window (activated or deactivated), notifying the object when it needs to resize its borders, and managing modeless dialog boxes.

The content object shows its toolbars and any modeless dialog boxes and calls COleServerDoc::OnResizeBorder(), which resizes and adjusts toolbars and other user-interface elements according to the size of the window.

Next, the content object calls the container's IOleInPlaceFrame::SetMenu() function to set the in-place menu inside the container.

Before actually making the content object visible, the server calls the container's IOleClientSite::ShowObject() function. This lands in COleClientItem::XOleClientSite::ShowObject() on the container's side, which calls COleClientSite::ShowObject(). ShowObject() makes sure the in-place item is showing in the container window.

The last operation performed by COleServerDoc::ActivateInPlace() is showing the in-place frame by calling the frame's ShowWindow() function.

In-Place Activation Epilogue

Whew—you made it! So that's how MFC does the in-place activation stuff! Hopefully, you can now fully appreciate all the work that MFC does for you. All that code would have been a nightmare to program by yourself.

Before leaving the topic of OLE documents, we need to talk a bit about (1) what happens when an item is deactivated and (2) how the Structured Storage is integrated with MFC's serialization model.

Deactivating the Item

An in-place active item is usually deactivated by clicking outside its bounds. An MFC container handles deactivating in-place active items by calling the active item's Close() function. This is interesting to watch because it will lead us to see how OLE's Structured Storage model and MFC's serialization mechanism work together.

One of the coolest things about MFC's support for OLE document servers is that as long as you buy into MFC's serialization model, the functionality for saving data to a container's file comes for free. That is, if you make your document use MFC's standard serialization mechanism you don't have to write any code to implement saving the document's COleClientItem objects.

Recall how MFC's serialization mechanism works. The framework calls your document's Serialize() function, in effect saying, "Time to save the data. What do you want to do about it?" MFC passes a pointer to a CArchive object. MFC just replaces the archive depending upon the mode. For example, if the server is running in standalone mode, the archive is based on a file created and/or opened by the application. If the server is running embedded in some other container application, the archive is based on an OLE Structured Storage object provided by the container. The application need not concern itself with whether it's running embedded or standalone—MFC hands over an archive based on the appropriate storage object. Let's follow the action as the client deactivates the active item by calling COleClientItem::Close().

Inside COleClientItem::Close()

COleClientItem::Close() changes the state of an OLE item from running to loaded. That is, the item's handler remains loaded in memory but the server is not running. An OLE container calls Close() with one of three options:

- OLECLOSE_SAVEIFDIRTY—Save the OLE item.
- OLECLOSE_NOSAVE—Do not save the OLE item.
- OLECLOSE_PROMPTSAVE—Prompt the user about whether to save the OLE item.

The first order of business is to call Close() through the content object's IOleObject interface. After making this call, control ends up in COleServerDoc::XOleObject::Close() on the server side. COleServerDoc::XOleObject::Close() simply delegates to the document's OnClose() function. If the embedding is dirty (that is, it's been changed), OnClose() calls COleServerDoc::SaveEmbedding().

COleServerDoc::SaveEmbedding()

COleServerDoc::SaveEmbedding() does just that: it saves an embedding in the document. To save the embedding, the server calls the container's IOleClientSite::SaveObject() function. Back in the container, this call lands in COleClientItem::XOleClientSite::SaveObject(). The client does a QueryInterface() for the content object's IPersistStorage interface through the IOleObject interface it's holding. The client then asks the object if it's dirty through IPersistStorage::IsDirty(). On the server side, the content object simply checks the document-dirty flag (you know, the one that is set by the function CDocument::SetModifiedFlag()).

If the content object is dirty, the client responds by calling the OLE API function OleSave() to save the content object to the container's storage. Here's the prototype:

```
HRESULT OleSave(
    IPersistStorage * pPS,
    IStorage * pStg,
    BOOL fSameAsLoad
);
```

OleSave() takes three parameters: (1) an IPersistStorage that belongs to the object being saved, (2) an IStorage pointer to hold the data, and (3) a flag indicating whether the object was loaded from the IStorage interface or not. COleServerDoc::SaveEmbedding() passes the content object's IPersistStorage pointer, a storage into which to place the content object's data, and a flag indicating whether it's going into the same storage from which it was loaded.

OleSave() performs the following steps:

1. Calls IPersistStorage::GetClassID() to get the CLSID.
2. Writes the CLSID to the storage object using WriteClassStg().
3. Calls IPersistStorage::Save() to save the object.
4. If there were no errors on the save, calls IPersistStorage::Commit() to commit the changes.

Once the container calls OleSave(), the very first stop is COleServerDoc::XPersistStorage::GetClassID(). GetClassID() just gets the content object's class ID from the class factory.

The next stop is COleServerDoc::XPersistStorage::Save(), which calls COleServerDoc::OnSaveEmbedding(). COleServerDoc::OnSaveEmbedding() then calls COleServerDoc::OnSaveDocument(), which then in turn calls COleLinkingDoc::OnSaveDocument(), which then in turn calls COleDocument::OnSaveDocument(). If the document is being saved to a storage

(that is, if the document's `m_lpRootStorage` is not `NULL`), then `COleDocument::OnSaveDocument()` calls the helper function `SaveToStorage()`. This lands in `COleLinkingDoc::SaveToStorage()`, which saves the object's class ID and writes it to the storage using the OLE API function `WriteClassStg()`. `COleLinkingDoc::SaveToStorage()` calls the base class's implementation, `COleDocument::SaveToStorage()`.

Here's where MFC creates an archive out of an OLE Structured Storage interface. `COleDocument::SaveToStorage()` declares a `COleStreamFile` object on the stack. `COleStreamFile` is a handy class that wraps an OLE `IStream` pointer. The key is that `COleStreamFile` objects are manipulated exactly like `CFile` objects. (That means you can create a `CArchive` object out of them—a key feature.) Once `SaveToStorage()` declares the `COleStreamFile`, it opens the stream file using `COleStreamFile::CreateStream()`. Once the `COleStreamFile` creates the stream, it can be used to construct a `CArchive` object.

And that's just what `SaveToStorage()` does: it creates a `CArchive` object out of a stream of the content object's substORAGE. Then the document goes through its normal serialization process. The document is blissfully unaware of the nature of the archive. It's actually very clever!

That's how the content objects save their data. Let's take a look at the container's document.

Saving the Container's Document

So, the content object saves its data to substORAGE whenever it closes (usually through in-place deactivation). However, there's still the matter of saving the container's document.

While OLE document content objects are responsible for storing their own data, OLE document containers are responsible for storing information *about* their collections of content objects. Let's follow the serialization process and examine this.

Of course, OLE document containers follow the same serialization mechanism as normal MFC documents. Somehow the user signals the application to save the document (usually by selecting Save from the File menu). This results in a call to the document's `OnSaveDocument()` function.

As with the content object, `OnSaveDocument()` creates a stream off the document's root storage and constructs a `CArchive` object based on that stream. Then `OnSaveDocument()` calls the helper function `SaveToStorage()`. `COleDocument::SaveToStorage()` goes ahead and saves the document's contents.

In addition to saving the document's native contents (for example, Scribble saves a list of stroke objects), the document is also responsible for saving information about its collection of COleClientItems. MFC's default implementation of COleDocument::Serialize() performs two functions: (1) it stores the number of COleClientItems in the CArchive object (which is really an OLE Structured Storage object), and (2) it walks the list of items, telling each item to serialize itself. So, if you have any special state information included with each item in the container (a rectangle position, perhaps), that information comes back when you load the file. In addition to any special state information, each COleClientItem has some data it needs to save: (1) the item number, (2) the display aspect, (3) the flag indicating whether to create a moniker upon loading the item, and (4) the current default drawing aspect. Once these items are written to the storage, COleClientItem::Serialize() calls COleClientItem::WriteItem(), which in turn commits the items to the storage.

Loading OLE Documents

Loading OLE documents is similar to saving them—the process just happens in reverse. In an MFC application, loading OLE documents is integrated into the serialization model as well.

When the user selects Open from the File menu, MFC routes the message to CWinApp::OpenDocumentFile(), which eventually culminates in a call to the document's OnOpenDocument() member. Because the document is derived from COleDocument, the framework calls COleDocument::OnOpenDocument(). COleDocument::OnOpenDocument() uses the OLE API function StgOpenStorage() to open the document. Like StgCreateDocFile(), StgOpenStorage() hands an IStorage pointer back to the caller. OnOpenDocument() sets the root storage member (*m_lpRootStorage*) to the newly opened storage. Now the document has a live IStorage pointer it can use.

COleDocument::OnOpenDocument() uses a helper function called LoadFromStorage() to read the items in the document. LoadFromStorage() basically performs the reverse of SaveToStorage(). Like SaveToStorage(), LoadFromStorage() creates a COleStreamFile object based on the document's root storage. Remember, you can use COleStreamFile to create CArchive objects—which is just what MFC's serialization model needs.

Next, OnOpenDocument() loads the document by calling the document's Serialize() function using the CArchive object (which is really an IStream pointer in disguise). After loading all the application-specific data (for example, if this were a word processor that supports OLE documents, the document would load all the word processing data), the document's Serialize() function calls COleDocument::Serialize() to load the client items.

MFC saves a count of the number of client items in the document whenever it serializes a COleDocument. Now the COleDocument needs that number to load the client items. COleDocument::Serialize() just serializes each item:

```
// read number of items in the file
DWORD dwCount;
ar >> dwCount;

// read all of them into the list
while (dwCount--) {
    CDocItem* pDocItem;
    ar >> pDocItem;      // as they are serialized, they are added!
}
```

Of course, MFC calls each item's Serialize() function as it is serialized. The client item first calls the base class's Serialize():

```
COleClientItem::Serialize(ar)
```

COleClientItem::Serialize() calls CDocItem::Serialize() to attach the item to the document. In addition, the item has to read in its member variables that it saved earlier: the item number, the cached display aspect, the moniker flag, and the default display aspect. Then COleClientItem::Serialize() calls ReadItem() to load the OLE object. ReadItem() opens the client item's substORAGE (remember, the storage that was created when the item was inserted) and assigns the IStorage pointer to COleClientItem::m_lpStorage. Then ReadItem() calls OleLoad() to bring the content object into the loaded state. OleLoad() performs the following functions:

1. Performs an automatic conversion of the object (if specified in the registry).
2. Gets the CLSID from the open storage object by calling IStorage::Stat().
3. Calls CoCreateInstance() to create an instance of the handler. By default, MFC uses the default handler—OLE32.DLL.
4. Passes the object a client site pointer by calling IOleObject::SetClientSite().
5. Performs a QueryInterface() for the object's IPersistStorage. If successful, OleLoad() calls IPersistStorage::Load().
6. Performs a QueryInterface() for the requested interface and returns that interface.

COleClientItem::ReadItem() asks for the object's IUnknown pointer and then uses it to QueryInterface() for the content object's IOleObjectPointer. ReadItem() saves this pointer in COleClientItem::m_lpObject.

COleClientItem::Serialize() finishes connecting to the client item by calling COleClientItem::FinishCreate().

Finally, if the item was embedded instead of linked, COleClientItem::Serialize() tries to get the object's IMoniker interface.

The document loads each client item this way until the whole document is loaded (and each content object is in a loaded state). When users double-click on the item or invoke some other verb, the object's IOleObject is available to do the container's bidding.

Conclusion

At one point OLE was all about compound documents. The old protocol was based upon dynamic data exchange, which had some serious performance and flexibility problems. Now that OLE documents are based on a set of COM interfaces, most of those problems have disappeared.

Like the other OLE technologies, OLE documents exists largely as a specification. MFC is just one way to implement them. However, if you decide to write your own implementation of the OLE document interfaces, you'll find yourself duplicating most of the work already done by the folks at Microsoft.

OLE documents and MFC's support for them could actually take up a whole book. We've touched upon the highlights here—their most interesting aspects. However, we didn't cover some things. For example, we didn't spend a whole lot of time on linking or monikers. To find out more, you may want to look in OLESVR1.CPP, OLESVR2.CPP, OLECLI1.CPP, OLECLI2.CPP, OLECLI3.CPP, OLEDOC1.CPP, and OLEDOC2.CPP. There's a lot of interesting stuff going on in there.

MFC and Automation

Automation is an OLE feature that continues to attract more and more attention as software developers realize its utility. You may be able to ignore it for the time being, but pretty soon, you won't be able to escape it. We'll tackle MFC and Automation in two parts: The first part covers OLE Automation in its raw state—what's happening at the COM level with an interface called IDispatch. The second part takes a look at how MFC implements Automation.

Understanding Automation at the COM level is important because it helps you realize that IDispatch (the main interface for dealing with Automation) is just another COM interface and that by implementing it, you add the feature known as Automation to your object.

The History of Automation

Automation grew out of the desire (1) to automate applications using tools that existed outside the application and (2) to integrate several applications at a high level using various common tools such as Visual Basic. In the Dark Ages (pre-OLE), many applications opened themselves to automation (not OLE Automation) and integration by one of two methods: (1) macro languages or (2) dynamic data exchange (DDE).

First, many applications provided their own macro languages. If you're a seasoned veteran, you may recall how each of the major spreadsheets (Lotus 1-2-3, Borland Quattro, and Microsoft Excel) maintained its own macro languages, which were very useful within the context of the application but were not very useful outside it. Though these macro languages made the applications automatable, they were generally confined to the application at hand and could not be used easily to drive more than one application at a time.

You may also recall that archaic data exchange technology called dynamic data exchange for sharing data between applications and for actually calling functions inside another application. Though DDE still exists, it's a slow, cumbersome protocol that's fairly difficult to implement.

What Automation Can Do for You

Automation is the OLE feature for solving the problems of proliferating macro languages and cumbersome data exchange protocols like DDE. However, Automation goes farther than simply automating applications. It's a useful technology that you can employ in many other places. For example, Automation is a key technology used by OLE controls. Automation is one of those features that may seem complicated or impractical at first. However, once you see it in action, you'll be sold on its utility and want to start implementing Automation everywhere you can.

Automation-enabled applications expose properties and methods to clients that are Automation savvy (like Visual Basic). For example, if you're programming a Visual Basic application that's an Automation client, it's very easy to access the application's Automation properties and functions. Imagine that someone has written an Automation server for you, called "StockMaster," that retrieves the current price of a specific stock on the New York Stock Exchange. That software might have methods that start and stop the acquisition, as well as some number representing the current price of the stock. In "Automation-ese," you say the object *exposes* two methods (starting and stopping the acquisition) and a property (the current price of the stock).

It's very easy to write a Visual Basic program to use such an object. In the [general] section of the VB program, you simply declare the object:

```
dim StockInfo as object
```

To bring the Automation object into being, you would simply place this line inside the "Form" section of your Visual Basic program:

```
StockInfo = CreateObject("StockMaster")
```

Then you add this line to the section of your program that initiates data collection:

```
StockInfo.StartAcquisition
```

And you add this line to the section of your program that stops data collection:

```
StockInfo.StopAcquisition
```

Finally, anytime you want to see the value of the current stock, you just peek at the property representing the current price:

```
StockInfo.CurrentPrice
```

Nothing could be easier!

Imagine how beneficial this can be. When you think about it, a lot of low-level effort would have to go into making something like StockMaster work. You'd need to agree on some communications protocols for receiving the data. Then you'd have to rig up some gunky RS-232 communications programming to establish a connection to the New York Stock Exchange. And on top of that, you'd have to write the code for managing the connection and monitoring data as it comes through. Arrgh. Automation is a way to abstract all the low-level details behind a programmable interface that scripting languages like Visual Basic can deal with.

Automation lets you treat whole programs as if they were objects. Automation presents a nice, even programming interface to Automation clients. Notice that to call StockMaster's functions, the Visual Basic client has only to dereference the function using a period (StockInfo.StartAcquisition). The Visual Basic program can access a property the same way: by dereferencing the property with a period (StockInfo.CurrentPrice).

Here's a good way to compare COM interfaces and dispatch interfaces. Dealing with COM interfaces is the anal-retentive way of working with COM objects; Automation is the relaxed way of working with COM objects. Languages like Visual Basic are much more relaxed about things like type checking and parameter passing. A Visual Basic program can take some parameters (it doesn't matter what type they are) and send them across the line to an Automation object. Once the call crosses over to the Automation object, it becomes the object's job to unpack the parameters, figure out what type they are, determine whether they make sense in the context of the function call, and finally set them up in an anal-retentive, C++-oriented stack frame so that the function call can be made.

When you see this kind of technology in action, you suddenly realize the avenues open to you for integrating applications. All of a sudden you can write software modules for a huge software audience (there are *tons* more VB programmers than there are Visual C++ developers—just stop by your local MIS shop to find out). If you want a way to make your software available to a huge software development audience, you should definitely check out Automation.

Writing an MFC Automation Application

If you've decided to buy into MFC, Automation is quite an easy feature to add—especially if you're starting from scratch. Just push the "Automation—Yes, please" button when using AppWizard. (Microsoft finally changed the label for the "No" button. Now it says, "Automation—No, thank you".) Even if you're not starting from scratch, adding Automation to an existing application is not a big deal. For more information, see Mary Kirtland's article on OLE Automation in the premiere issue of *VC++ Professional* (January 1995). From there, ClassWizard supplies all the support you need to add Automation properties and methods to your application.

But How Does It All Work?

Automation in action really is amazing. The word *magical* comes to mind. But how can you get two applications to talk together so easily?

As with all OLE features, Automation works through the magic of the Component Object Model (COM). COM is a lifestyle—a way of programming. Automation lies on top of COM, implemented through a specific COM interface called *IDispatch*.

One of the more confusing aspects of COM and OLE is that COM and OLE are largely specifications describing how to implement certain compelling features into your app. About 15% of the functionality defined by OLE has been given a default implementation by Microsoft. For example, you can acquire default implementations of Structured Storage services and memory-allocation services by calling some global functions provided by Microsoft (for example, call `StgCreateDocFile()` to retrieve Structured Storage services and `CoGetMalloc()` to retrieve OLE memory-management services).

However, the other 85% of the services defined by Microsoft are not implemented at all. It's up to software developers around the world to implement these features themselves. One of the services that doesn't have a default implementation is Automation. That's because Automation really exists within the domain of applications (as opposed to operating systems). So, if you want to add Automation to your application, you have to implement it yourself (or find someone else to implement it for you—perhaps a framework).

COM Interfaces versus Automation

Before diving into the murky waters of Automation, let's quickly review the difference between COM interfaces and Automation. After all, both are really just ways of allowing multiple applications to share functionality.

COM Interfaces Reviewed

Recall that in COM, all functionality is provided via COM interfaces. COM interfaces are really just bundles of functions that define a service. As you saw in Chapter 11, OLE memory management happens through a COM interface. The OLE function CoGetMalloc() retrieves a pointer to the *task allocator*. By retrieving an IMalloc pointer, you all of a sudden have nine functions you can use through the interface. These functions include QueryInterface(), AddRef(), Release(), Alloc(), Free(), Realloc(), GetSize(), DidAlloc(), and HeapMinimize(). Using COM's call-use-release protocol, you *call* CoGetMalloc() to retrieve the interface pointer, you *use* the interface functions, and you *release* the interface when done. The following code snippet shows how this works. The function AllocMem() acquires an IMalloc pointer using the OLE function CoGetMalloc(). It uses IMalloc to allocate and then free a piece of memory.

```
HRESULT AllocMem() {
    LPMALLOC pMalloc;
    LPSTR lpsz;

    ::CoGetMalloc(MEMCTX_TASK, &pMalloc);
    if (lpsz = LPSTR(pMalloc->Alloc(256)))
        result = NOERROR;
    else
        result = ResultFromScode(E_OUTOFMEMORY);

    if(lpsz)
        pMalloc->Free(lpsz);

    pMalloc->Release();

    return result;
}
```

So, an interface is really nothing but a function table that OLE clients can access. Recall IMath from Chapter 11: IMath is a custom interface that supports two functions (above and beyond IUnknown)—Add() and Subtract(). Admittedly, this is a fairly simple interface. However, it's small enough and clean enough to demonstrate

easily the differences between COM interfaces and Automation. Here's the interface definition for IMath:

```
interface IMath
{
    // *** IUnknown methods ***
    virtual HRESULT QueryInterface(REFIID riid, LPVOID FAR* ppvObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;

    // *** IMath methods ***
    virtual HRESULT Add(INT, INT, LPLONG) = 0;
    virtual HRESULT Subtract(INT, INT, LPLONG) = 0;
};
```

Any COM object can implement this interface (along with other interfaces, perhaps). Chapter 11 contains a description of a COM class called CoMath that implements this interface.

To use this interface, a COM client just needs to create an instance of the object and use the interface. Here's the code that does the trick:

```
{
...
IMath *pMath;
HRESULT hr;

hr = ::CoCreateInstance(CLSID_CoMath, NULL,
                        CLSCTX_INPROC_SERVER,
                        IID_IMath,
                        (VOID FAR **)&pMath);

LONG lSum, lDifference;

hr = pMath->Add(10, 50, &lSum);
hr = pMath->Subtract(150, 50, &lDifference);
...
}
```

Because the interface is defined in a header file that you include in your source code when you compile a client program, the client program is aware of the structure of the interface at compile time. Once you create an instance of an object implementing IMath and retrieve an IMath interface pointer, you can call the IMath functions as much as you like. When the compiler hits any of the IMath function calls, the compiler knows how to resolve the function call based on its offset into the interface vtable. Figure 14-1 illustrates this setup.

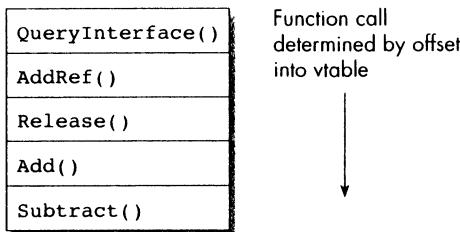


Figure 14-1. The IMath interface

The main point here is that COM interfaces use *early binding*. That is, the compiler knows ahead of time (before run time) about the interface's vtable, as well as the signatures for each of the functions within the interface. This is perfect for any compiled language that uses early binding. However, recall that there are a lot of people out there using interpreted languages like Visual Basic (and that they have checkboxes). This scheme falls apart because interpreted languages require function call binding at run time (a.k.a. *late binding*). Enter Automation and a new interface called IDispatch.

IDispatch: The Key to Automation

Automation requires some sort of *late binding* mechanism because the clients for which Automation was first intended are unable to deal with the standard, early binding mechanism of regular COM interfaces. Clients that use normal COM interfaces employ strict typing, understand all the parameters ahead of time, and set up the stack frame perfectly before the function call is made. Automation works a little bit differently.

Imagine an Automation client that wants to call the function Foo() from an Automation server. Because interpreted languages are generally much more relaxed programming environments than C++, automation takes a much more laid-back approach to sharing functionality between applications. With Automation, the client says, “Okay, I think there’s a function named Foo() on the other end, and I have some parameters. Is there something you can do for me, Mr. Object?” Then it’s up to the object (1) to determine if there is indeed a function called Foo(), (2) to evaluate the parameters, (3) to find out if the parameters make sense, and (4) to make the function call. In OLE, IDispatch is the magic interface that does all this work. IDispatch has four functions (which you’ll see shortly). Of the four, Invoke() is the function that does the late binding work.

Now imagine a COM object called DispMath that implements the IDispatch interface so that a client such as Visual Basic could perform late binding on the Add() and Subtract() functions. Figure 14-2 illustrates this relationship.

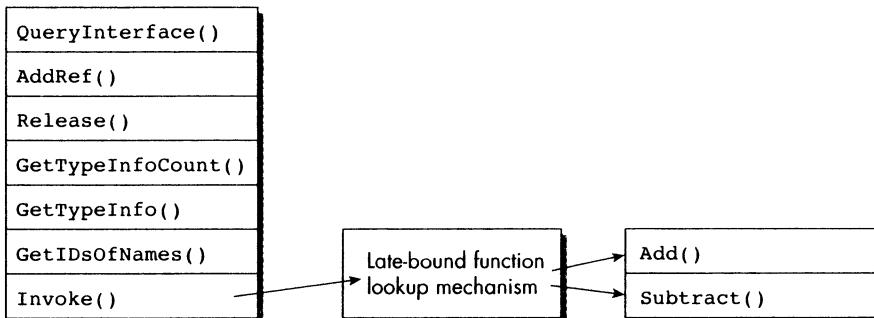


Figure 14-2. IDispatch (implemented by DispMath)

The figure illustrates how IDispatch is just another COM interface. It's got the three standard IUnknown functions on top, followed by the IDispatch functions. However, instead of being a custom vtable that implements Add() and Subtract(), DispMath's interface uses a (magical, at this point) mechanism to manage parameter passing and to resolve the function pointers.

Under this scenario, a client may query DispMath for IDispatch. Because DispMath implements IDispatch, DispMath returns an IDispatch pointer to the client. Once the client has a pointer to IDispatch, it can call IDispatch::Invoke() to make the function call.

Before examining the Invoke() mechanism, let's take a closer look at IDispatch. Here's IDispatch's definition:

```

interface IDispatch : IUnknown {
    virtual HRESULT GetTypeInfoCount(unsigned int *pcTInfo) = 0;
    virtual HRESULT GetTypeInfo(unsigned int iTInfo, LCID lcid,
                               ITypeInfo *ppTInfo) = 0;
    virtual HRESULT GetIDsOfNames(REFIID riid, OLECHAR **rgszNames,
                                 unsigned int cNames, LCID lcid,
                                 DISPID *rgdispid) = 0;
    virtual HRESULT Invoke(DISPID dispID, REFIID riid, LCID lcid,
                          unsigned short wFlags,
                          DISPPARAMS *pDispParams,
                          VARIANT *pVarResult, EXCEPINFO *pExcepInfo,
                          unsigned int *puArgErr) = 0;
};

```

Notice that IDispatch has four functions: GetTypeInfoCount(), GetTypeInfo(), GetIDsOfNames(), and Invoke(). As far as MFC goes, GetIDsOfNames() and Invoke() are the most important functions. (In fact, until recently, MFC implemented neither

`GetTypeInfo()` nor `GetTypeInfoCount()`.) Let's start by examining `IDispatch::Invoke()` and `IDispatch::GetIDsOfNames()`. Then we'll examine type information.

`IDispatch::Invoke()`

Take a look at `IDispatch::Invoke()`. This is the backbone of Automation. `Invoke()` isn't for the light-hearted—it accepts eight (count 'em, eight) parameters! The first parameter, `dispid`, identifies the Automation method or property to invoke. `DISPID` is a 32-bit magic cookie assigned by the Automation server to identify its properties and methods. The dispatch ID is similar to a system handle in that as a developer, you don't really care what the value is—you simply need to obtain one so that you can talk intelligently to the system. More about `DISPIDs` in a minute.

The second parameter is a reference to an interface ID. This parameter is currently reserved for future use and should always be `IID_NULL`. The third parameter, `lcid`, represents the locale context in which to interpret arguments. Applications that don't support multiple national languages can ignore this parameter.

Because `Invoke()` exposes both methods and properties, `Invoke()` needs some way of distinguishing the context of the call. The fourth parameter, `wFlags`, communicates the context of the call. This can be one of four values: `DISPATCH_METHOD`, `DISPATCH_PROPERTYGET`, `DISPATCH_PROPERTYPUT`, or `DISPATCH_PROPERTYPUTREF`. A value of `DISPATCH_METHOD` in this field tells `Invoke()` that the client is trying to make a function call. A value of `DISPATCH_PROPERTYGET` or `DISPATCH_PROPERTYPUT` indicates that the client is trying to retrieve or set, respectively, an Automation property. Finally, `DISPATCH_PROPERTYPUTREF` means that the client is trying to change a property via a reference assignment rather than a value assignment.

Of course, the ability to pass parameters to functions is very important, regardless of whether or not the function calls are late bound or early bound. That's what the fifth `Invoke()` parameter represents. The `DISPPARAMS` structure describes a variable list of arguments represented by another structure, called a `VARIANT`. Here's the `DISPPARAMS` structure definition:

```
typedef struct FARSTRUCT tagDISPPARAMS{
    VARIANTARG FAR* rgvarg;           // Array of arguments
    DISPID FAR* rgdispidNamedArgs;   // Dispatch IDs of named arguments
    unsigned int cArgs;              // Number of arguments
    unsigned int cNamedArgs;         // Number of named arguments
} DISPPARAMS;
```

Automation depends on binding functions at run-time as opposed to at compile time. That also means that data isn't known ahead of time. For that reason, `IDispatch` uses a self-describing data type called a `VARIANT`. The first field of the `VARIANT`

structure is an unsigned short representing the type of data contained in the structure. A VARIANT can represent any of these 25 data formats, as shown in Listing 14-1.

Listing 14-1. The VARIANT data type

```
typedef struct tagVARIANT {
    VARTYPE vt;
    unsigned short wReserved1;
    unsigned short wReserved2;
    unsigned short wReserved3;
    union {
        short      iVal;           /* VT_I2             */
        long       lVal;           /* VT_I4             */
        float      fltVal;         /* VT_R4             */
        double     dblVal;         /* VT_R8             */
        VARIANT_BOOL bool;         /* VT_BOOL            */
        SCODE      scode;          /* VT_ERROR           */
        CY         cyVal;          /* VT_CY              */
        DATE       date;           /* VT_DATE             */
        BSTR       bstrVal;         /* VT_BSTR             */
        IUnknown   FAR* punkVal;    /* VT_UNKNOWN           */
        IDispatch  FAR* pdispVal;   /* VT_DISPATCH           */
        SAFEARRAY   FAR* parray;     /* VT_ARRAY|*           */
        short      FAR* piVal;       /* VT_BYREF|VT_I2           */
        long       FAR* plVal;       /* VT_BYREF|VT_I4           */
        float      FAR* pfltVal;    /* VT_BYREF|VT_R4           */
        double     FAR* pdblVal;    /* VT_BYREF|VT_R8           */
        VARIANT_BOOL FAR* pbool;    /* VT_BYREF|VT_BOOL           */
        SCODE      FAR* pscode;      /* VT_BYREF|VT_ERROR           */
        CY         FAR* pcyVal;      /* VT_BYREF|VT_CY              */
        DATE       FAR* pdate;        /* VT_BYREF|VT_DATE             */
        BSTR       FAR* pbstrVal;    /* VT_BYREF|VT_BSTR             */
        IUnknown   FAR* FAR* ppunkVal; /* VT_BYREF|VT_UNKNOWN           */
        IDispatch  FAR* FAR* ppdispVal; /* VT_BYREF|VT_DISPATCH           */
        SAFEARRAY   FAR* FAR* parray;  /* VT_ARRAY|*           */
        VARIANT    FAR* pvarVal;     /* VT_BYREF|VT_VARIANT           */
        void       FAR* byref;        /* Generic ByRef           */
    };
};
```

Using VARIANTs makes passing data around fairly simple because all the parameters are homogenous. However, that does mean a bit more work for both the Automation client and the Automation server. The Automation client has to package its parameters into VARIANTs and the server has to unwrap them. The server also has

to wrap the out parameters in a VARIANT, and the client has to unpack any out parameters. Fortunately, there are a series of OLE functions available for coercing VARIANTs to and from various types.

The sixth parameter represents the result of the function call. It's also packaged in a VARIANT.

The last two parameters handle error conditions and error codes. OLE defines an exception structure that `Invoke()` can fill in the event of an exception (for example, the client used an invalid DISPID).

At this point, there's only one missing piece to the IDispatch puzzle. Notice that the only way `Invoke()` knows what function or property to access is through the DISPID parameter. Yet, a Visual Basic program uses human-readable names to access the functions. How can you get the dispatch ID if you're given only a human-readable name? IDispatch has a function for just that purpose: `GetIDsOfNames()`.

IDispatch::GetIDsOfNames()

Automation clients can use `IDispatch::GetIDsOfNames()` to figure out a DISPID if given a human-readable name. `GetIDsOfNames()` maps a dispatch member (a property or a method) and an optional set of argument names to a corresponding set of integer DISPIIDs. That is, feed `GetIDsOfNames()` the human-readable name for which you would like the DISPID (along with any argument names) and an empty array of DISPIIDs. `GetIDsOfNames()` fills the DISPID array with corresponding values, and you may use the DISPIIDs to call `IDispatch::Invoke()`.

Implementing IDispatch by Hand

The two most important IDispatch functions to implement are `Invoke()` and `GetIDsOfNames()`. There are several issues you need to face to do so. You need to perform several steps:

1. Make up DISPIIDs for your methods and properties.
2. Create some sort of mechanism to map DISPIIDs to functions and properties within your code.
3. Implement `Invoke()` to use the mapping mechanism to allow clients to call methods and access properties.
4. Implement `GetIDsOfNames()` so the client can look up your DISPIIDs given some human-readable names.

Choosing DISPIIDs is more or less up to you within certain constraints. Remember, DISPIIDs are simply magic cookies that clients use to invoke functions and access

properties. However, Microsoft has reserved all negative DISPID values for future use. So far, Microsoft has defined the standard DISPIIDs as shown in Table 14-1.

Table 14-1. Standard DISPIIDs

Symbol	Value	Meaning
DISPID_VALUE	0	Indicates the default member for the dispatch interface.
DISPID_UNKNOWN	-1	Returned by GetIDsOfNames() to indicate that a dispatch ID couldn't be found.
DISPID_PROPERTYPUT	-3	Indicates the parameter that receives the value of an assignment when setting an OLE automation property.
DISPID_NEWENUM	-4	Used for collections.
DISPID_EVALUATE	-5	Used by the controller implicitly to evaluate expressions within square brackets.
DISPID_CONSTRUCTOR	-6	Identifies a method that acts as the object's constructor. Reserved.
DISPID_DESTRUCTOR	-7	Identifies a method that acts as the object's destructor. Reserved.

In addition to these seven values, Microsoft has reserved the DISPID range between -500 and -999 for OLE controls. Other than that, you're completely free to use the numbers between one and two billion. That's a lot of DISPIIDs to choose from.

Implementing Invoke() by Hand

Though implementing Invoke() is fairly straightforward, it can be fairly cumbersome. First, for each dispatch method you want to create, you need to write the C++ function that performs the work. Then, for each dispatch property you create, you need to add a variable to your C++ code to hold the value. That's no big deal—you would have had to do that anyway.

Next, you need to devise some system of mapping incoming dispatch IDs to your C++ code. This could be as simple as writing a switch statement that switches on each of your known dispatch IDs or it could be as elaborate as a well-optimized hash table. Any way you do it, you have to map the incoming DISPID to some real code within your program.

Invoke() has a fairly complex signature. In addition to the DISPID, Invoke() supplies a great deal of other information useful for implementing it. If you decide to deal with locales, then you have to pay attention to the lcid parameter. You need to use the wFlags parameter to find out if the client is trying to call a method or

access a property. However, the most complex parameters are those that pass parameters and results back and forth.

Fun with VARIANTS

Dealing with the pDispParams and pVarResult parameters can be somewhat tedious. Remember that IDispatch parameters are passed and results are returned within VARIANTS. Invoke() doesn't deal with a well-known parameter list understood at compile time. All this stuff happens at run time. It's up to you to go through the list of VARIANT parameters, unpack each one and convert them to their native representations, and return any results in a VARIANT structure. Fortunately, Microsoft includes a number of functions and macros for working with VARIANTS: VariantInit(), VariantClear(), and VariantChangeType().

VariantInit() initializes a VARIANT structure. VariantClear() empties out the VARIANT structure and should be used before the VARIANT goes out of scope. Finally, VariantChangeType() coerces a VARIANT from one type to another. This is useful for making sure the parameters passed in are of the correct type.

Microsoft also provides a number of extraction macros that follow this pattern:

```
V_<some type> = <native data>
```

For example, if you want to insert a long value called “lSomeValue” into an instance of a VARIANT called “var”, just use the following macros:

```
VT(&var) = VT_I4;
V_I4(&var) = lSomeValue;
```

If you'd like to extract a long value from the same VARIANT structure, you could use the same macro, like this:

```
lSomeValue = V_I4(var);
```

OLE provides a function called DispGetParam() that fetches a specific VARIANT parameter from a DISPPARAMS array: this function takes your DISPPARAMS array and an index into that array and fills another VARIANT structure with the specified parameter.

Finishing Invoke()

The final two parameters deal with exceptions and error codes. If the client is interested in exception information, the client passes an EXCEPINFO structure that you fill with appropriate information. Finally, the client passes a pointer to an unsigned integer, which you can fill with an error code.

All in all, Microsoft provides a good number of tools to implement IDispatch::Invoke(). However, doing this by hand is not without its perils. Implementing Invoke() for the DispMath object might look something like Listing 14-2.

Listing 14-2. IDispatch::Invoke() implemented manually

```
STDMETHODIMP
DispMath::XDispObj::Invoke(DISPID dispIdMember, REFIID riid,
                           LCID lcid, WORD wFlags,
                           DISPPARAMS FAR *pdispparams,
                           VARIANT FAR *pvarResult,
                           EXCEPINFO FAR *pexcepinfo,
                           UINT FAR *puArgErr) {

    unsigned int uArgErr;
    VARIANTARG varg1;
    VARIANTARG varg2;
    VARIANT varResultDummy;

    // Make sure the wFlags are legal
    if(wFlags & ~(DISPATCH_METHOD | DISPATCH_PROPERTYGET |
                  DISPATCH_PROPERTYPUT | DISPATCH_PROPERTYPUTREF))
        return ResultFromScode(E_INVALIDARG);

    // This object only exposes a "default" interface.
    if(!IsEqualIID(riid, IID_NULL))
        return ResultFromScode(DISP_E_UNKNOWNINTERFACE);

    // It simplifies the following code if the caller
    // ignores the return value.
    if(puArgErr == NULL)
        puArgErr = &uArgErr;
    if(pvarResult == NULL)
        pvarResult = &varResultDummy;

    VariantInit(&varg1);
    VariantInit(&varg2);

    // Assume the return type is void, unless we find otherwise.
    VariantInit(pvarResult);

    unsigned long ulFirstNo, ulSecondNo;
    long lResult;
    unsigned int uError;
```

```

// Unpack the parameters
DispGetParam(pdispparams, 0, VT_I4,&varg1, &uError);

DispGetParam(pdispparams, 1, VT_I4, &varg2, &uError);

ulFirstNo = V_I4(&varg1);
ulSecondNo = V_I4(&varg2);

switch(dispidMember){

    case DISPID_ADD:
        lResult = ulFirstNo + ulSecondNo;
        V_VT(pvarResult) = VT_I4;
        V_I4(pvarResult) = lResult;
        break;

    case DISPID_SUBTRACT:
        lResult = ulFirstNo - ulSecondNo;
        V_VT(pvarResult) = VT_I4;
        V_I4(pvarResult) = lResult;
        break;

    default:
        return ResultFromScode(DISP_E_MEMBERNOTFOUND);
}

return NOERROR;
}

```

Notice that most of the code involves error checking and the pure mechanics of unpacking the parameters in a useful form. It's the server's job to make sure all the parameters are kosher and that the correct type of result is passed back.

Implementing GetIDsOfNames()

GetIDsOfNames() is somewhat easier to handle. All you really need to do is examine the list of dispatch function and property names that come in and fill the empty array of DISPIDs. Listing 14-3 is a minimal implementation of GetIDsOfNames() within the context of CoMath.

Listing 14-3. IDispatch::GetIDsOfNames() implemented manually

```

STDMETHODIMP
DispMath::XDispObj::GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames,
                                  UINT cNames, LCID lcid,
                                  DISPID FAR *rgdispid) {
    SCODE sc = S_OK;

```

```

// check arguments
if (riid != IID_NULL)
    return DISP_E_UNKNOWNINTERFACE;

// fill in the member name
if (lstrcmpi(rgszNames[0], "Add") == 0) {
    rgdispid[0] = DISPID_ADD;
} else if (lstrcmpi(rgszNames[0], "Subtract") == 0) {
    rgdispid[0] = DISPID_SUBTRACT;
} else {
    rgdispid[0] = DISPID_UNKNOWN;
    sc = DISP_E_UNKNOWNNAME;
}

// make the argument names DISPID_UNKNOWN for this implementation
for (UINT nIndex = 1; nIndex < cNames; nIndex++)
    rgdispid[nIndex] = DISPID_UNKNOWN;

return sc;
}

```

You may notice in the previous example that two IDispatch functions aren't fully implemented: GetTypeInfo() and GetTypeInfoCount(). In addition, you may be marveling at the complexity of the example. It takes a lot of work to pack and unpack the parameters properly and then look up the function and call it. The example shows one way of implementing IDispatch (it's actually the hard way). Perhaps this isn't the ideal way to implement IDispatch. In fact, there is an easier way—by using something called type information.

George's Automation Anecdote

One thing I do to earn my living is travel the countryside delivering classes on how to develop OLE applications using MFC. On several occasions, I've encountered an interesting situation. The company for whom I'm delivering the class may have ten developers working on the company's latest OLE endeavor. However, there are only nine people taking the class. When I ask, "Why isn't Jane/Joe taking the class?" the response is usually this: "Oh—he/she implemented our Automation interface for us a couple of years ago. He/she doesn't need this class. Yeah—Jane/Joe did it the hard way!" The person in question is usually a renegade who has earned a great deal of respect from the development team for tackling Automation without the help of MFC. Mind you, this has occurred several times now! It goes to show the importance of Automation in the grand scheme of OLE.

It's easy to forget that there's more than one way to implement Automation in your application and that using MFC's ClassWizard to do it is only one way. Using ClassWizard remains the fastest and most convenient way to get your Automation interface under way.

Another Way: Using Type Information

Type information is becoming one of the most important pieces of Automation. It's absolutely critical to OLE controls. Let's examine type information briefly; we'll see a lot more when we get to OLE controls.

Type Information

Type information is a way of advertising information about an object (including interfaces—dispatch interface or otherwise) to clients. For example, a client can use type information to learn about the object's interfaces without calling `QueryInterface()`. One of the best ways to think about type information is to look at it like a binary header file. For example, when you write functions or classes in one module and you want to use them in another, what do you do? You publish the information in a header file. The preprocessor understands the syntax of the header file and does some black magic like setting up symbol tables so the compiler can use the information to create executable code. Type information is the same idea, but within the context of COM interfaces.

As you can imagine, type information promises to be extremely important within the context of development tools based on the idea of browsing. If there's a way to find out an object's properties, methods, interfaces, and so forth ahead of time, that creates the possibility of writing development browsers that interrogate the objects beforehand, present information about the objects to the developer, and then let the developer link together programmable objects using some sort of browser.

But where does type information come from? Uh-oh—now there's another language to learn: *Object Description Language*.

Object Description Language

The easiest way to generate type information for your COM class is to write a script in Object Description Language (ODL) and compile it using a tool called `MKTYPLIB.EXE`. (There are other ways to create type information, but they're not worth mentioning here. It's just best to let Bill (that is, Microsoft) compile your type information using `MKTYPLIB`.) Object Description Language describes type information by first stating its attributes (things like GUIDs, versions, and locality information) then actually describing the type information.

For example, the ODL script for the Math object just described might look something like Listing 14-4.

Listing 14-4. ODL script describing CoMath

```
01 [
02     uuid(06812841-46e9-11ce-aeff-e798a8721416)
03     helpstring("MFC Internals CoMath 1.0 Type Library"),
04     lcid(0x9),
05     version(1.0)
06 ]
07 library Math08
08 {
09
10     importlib("stdole.tlb");
11
12     [
13         uuid(06812842-46e9-11ce-aeff-e798a8721416)
14         helpstring("IMath Dispatch Interface"),
15         odl
16     ]
17     interface IMathDisp : IUnknown
18     {
19         [propput, helpstring("Set the current value"), id(0)]
20         void CurrentValue([in] long val);
21
22         [propget, helpstring("Get the current value"), id(0)]
23         long CurrentValue(void);
24
25         long Add([in] long val1, [in] long val2);
26         long Subtract([in] long val1, [in] long val2);
27     }
28
29
30     [
31         uuid(06812843-46e9-11ce-aeff-e798a8721416)
32         helpstring("DMathDisp Dispatch Interface")
33     ]
34     dispinterface DMathDisp
35     {
36         interface IMathDisp;
37     }
38
39     [
40         uuid(06812840-46e9-11ce-aeff-e798a8721416)
41         helpstring("CoMath: A COM Math Class")
42     ]
43     coclass CoMath
44     {
45         dispinterface DMathDisp;
46         interface IMathDisp;
47     }
48 }
```

The first six lines of the script describe the attributes of the whole library. The type library has a GUID (man, those things are all over the place). The type library also contains a version number and some help information. Notice that this information is blocked off by the square brackets.

The lines following (lines 7–48) describe the type library itself. Notice that the type library has several blocks of type information. The type library includes type information about a specific set of interfaces (IMathDisp and DMathDisp) and the COM class that implements the interfaces.

Once the information about your COM object is planted inside a type library, you can use the type library to implement IDispatch.

Implementing IDispatch Using Type Information

If you've got your COM object and dispatch interfaces defined via type information, then there are a couple of other ways to have Bill (that is, Microsoft) implement IDispatch for you (rather than doing everything by hand). COM has two standard ways of using type information to implement IDispatch. However, to exploit that capability, you need to load the type library to retrieve the necessary interface pointers: ITypeLib and ITypeInfo. (That makes sense—we're in COM land now, where *everything* is done through interface pointers!)

Loading a Type Library

OLE has several API functions for loading type libraries. From the point of view of an Automation object, the two most common ones are LoadRegTypeLibrary() and LoadTypeLibrary(). LoadRegTypeLibrary() loads the type library from the registry and returns an ITypeLib pointer to the caller. LoadTypeLibrary() takes the path to the type library as the first parameter and returns an ITypeLib pointer to the caller. LoadTypeLibrary() has the nice side effect of loading the type library and making an entry into the registry (if the type library isn't already in the registry).

You may notice from the ODL file that the entire type library is identified by a GUID. In addition, each chunk of type information inside the type library is identified by a GUID. ITypeLib has a member function called GetTypeInfoOfGuid(). As long as you know the GUID of the specific block of type information you want, GetTypeInfoOfGuid() will get you an ITypeInfo pointer representing that block.

So now that you have an ITypeInfo pointer, what can you do with it?

Exploiting the Type Information

ITypeInfo is one of those gnarly COM interfaces, with some 19 functions (not counting the IUnknown functions)! Many of the functions are there for retrieving various aspects of the type information, such as information about functions and parameters.

You do not want to deal with most of these functions head-on (although there's nothing stopping you from doing so). However, `ITypeInfo` has two very interesting member functions: `Invoke()` and `GetIDsOfNames()`. Upon close examination, you'll notice that these functions are suspiciously similar to `IDispatch::Invoke()` and `IDispatch::GetIDsOfNames()`:

```
HRESULT ITypelibInfo::Invoke(void *lpvInstance
    DISPID dispidMember,
    WORD wFlags,
    DISPPARAMS FAR* pdispparams,
    VARIANT FAR* pvarResult,
    EXCEPINFO FAR* pexcepinfo,
    UINT FAR* puArgErr);

HRESULT ITypelibInfo::GetIDsOfNames(char FAR* FAR* rgszNames,
    UINT cNames,
    DISPID FAR* rgdispid);
```

The main difference is that first parameter to `Invoke()`. This parameter represents the COM class that implements the dispatch interface. So where does this class come from?

You (as the developer) have to write the class. However, MKTYPLIB will give you a great head start. One by-product of compiling type information with MKTYPLIB is that MKTYPLIB will give you a regular C header file defining an interface that maps to the type information in your ODL file. To exploit the type information, just write a COM class deriving from `IDispatch` and the interface defined in the header file. Be sure to load the type library (using `LoadTypeLibrary()`) and extract the type information (using `GetTypeInfoOfGuid()`) inside your COM class's constructor. Then when a client calls your COM object's `Invoke()` or `GetIDsOfNames()` members, just delegate to `ITypelibInfo`'s versions of the functions. Of course, the first parameter to `ITypelibInfo::Invoke()` is a pointer to your COM class. After that, OLE does all the rest of the work packaging the functions and resolving DISPIDs to real working functions.

IDispatch::GetTypeInfo() and IDispatch::GetTypeInfoCount()

Now that you know something about type information, `IDispatch::GetTypeInfo()` and `IDispatch::GetTypeInfoCount()` make a bit more sense.

If you've got a type library to describe your Automation interface, it's no big deal to make this information available to the Automation clients. Simply load the type library (using `LoadTypeLibrary()`), extract the type information (using `GetTypeInfoOfGuid()`), and return the `ITypelibInfo` pointer to the client.

To implement `GetTypeInfoCount()`, just return the number 1 to indicate that type information is available. Return 0 if there isn't any type information available.

One More Option: `CreateStdDispatch()`

But wait—there's one more way to implement `IDispatch`. You can use another COM API function called `CreateStdDispatch()`. Here's the prototype:

```
STDAPI CreateStdDispatch(
    IUnknown FAR* punkOuter,
    void FAR* pvThis,
    ITypeInfo FAR* ptinfo,
    IUnknown FAR* FAR* ppunkStdDisp);
```

When you hand your type information over to `CreateStdDispatch()`, the function creates a COM object that implements `IDispatch` according to the information represented by the `ITypeInfo` pointer. Notice that the first parameter is a pointer to a controlling `IUnknown`. Hmm... all that crazy stuff you learned about COM aggregation in Chapter 11 wasn't a waste after all! `CreateStdDispatch()` actually creates an aggregate object out of your object, the COM object generated by `CreateStdDispatch()`, and any other members hidden behind the controlling `IUnknown`. `CreateStdDispatch()` returns an `IDispatch` that you hand out to clients during `QueryInterface()`.

Recap: Automation in the Raw

`DispMath`'s first implementation of `IDispatch` isn't quite complete. It was missing type information. However, the first version of `DispMath` will work fine as a simple Automation object that Visual Basic (or another Automation client that knows what to do with `GetIDsOfNames()`) can use. The diskette that accompanies this book includes a program called `MATHSRVR`. It's a regular MFC app that houses the `DispMath` object. Use the Visual Basic program `TESTMATH.MAK` to exercise it. You can run `MATHSRVR` through the debugger to see the action take place.

The second and third ways to implement `IDispatch` involve generating a type library and exploiting the type information.

Why did we bring you through this detour? You're probably banging your fist on the table, saying, "I want my MFC!" One of the hurdles to fully understanding COM (and OLE) is that they are mostly specifications. There are often myriad ways to implement a single interface. `IDispatch` is a great example of that. Once you see Automation implemented in the raw, MFC's implementation makes a lot more sense. The next part shows how MFC does it. As with most OLE features implemented through MFC, Automation finds its roots in `CCmdTarget`. We'll examine the extensions to `CCmdTarget` and the dispatch map macros that MFC uses to make Automation possible.

MFC and Automation

Now that you've had a chance to see how Automation works at the COM level, let's check out how MFC does it. Recall that Automation happens through a single interface called `IDispatch`, and to support Automation, a COM class just has to implement `IDispatch`. `IDispatch`'s four member functions include `Invoke()`, `GetIDsOfNames()`, `GetTypeInfo()`, and `GetTypeInfoCount()`. Recall that `Invoke()` uses a 32-bit magic cookie called a `DISPID` to invoke Automation methods and Automation properties. Automation clients can obtain the `DISPIDs` for various human-readable names using `GetIDsOfNames()`. The other two functions, `GetTypeInfo()` and `GetTypeInfoCount()`, are used to access type information associated with the Automation object.

The program `MATHSRVR` housed an object called `DISPMATH` to illustrate Automation. `DISPMATH` provided a minimal implementation of `IDispatch`, fully implementing `Invoke()` and `GetIDsOfNames()` while stubbing out `GetTypeInfo()` and `GetTypeInfoCount()`. `DISPMATH`'s `Invoke()` function illustrated how to use `VARIANTs` to pass parameters using Automation.

In addition, there are other ways to implement `IDispatch` using type information.

But that's the hard way to do it. One of Visual C++'s main features is its support for Automation. AppWizard makes it easy to generate Automation-enabled applications, and ClassWizard makes it easy to add methods and properties to Automation objects. Behind the facade of dialog boxes, MFC does the work required to make `IDispatch` function. But how does MFC do it? What happens when you press those buttons? This section examines how MFC implements Automation. To find out, we'll look at extensions to `CCmdTarget` and a device called dispatch maps, which MFC uses to enable dynamic invocation (a.k.a. Automation).

Extensions to `CCmdTarget`

As with most of MFC's OLE support, Automation happens through `CCmdTarget`. That's because `CCmdTarget` implements `IUnknown` and `IDispatch`. One great advantage of this is that you can add Automation support to any class derived from `CCmdTarget`. So, if `CCmdTarget` is the key to Automation, `CCmdTarget` must implement `IDispatch`.

What? No Interface Map Entry for `IDispatch`?

MFC implements COM using the nested classes/composition approach. MFC has a set of macros (what else?) to set up the nested classes for each of the necessary interfaces. For example, MFC has a class called `COleDataSource` that implements the `IDataObject` interface. If you look for the definition of `COleDataSource` in

AFXOLE.H, you'll see that it uses the macros to declare a nested class for implementing IDataObject as well as an interface map (which CCmdTarget uses to implement QueryInterface()). So, since CCmdTarget implements IDispatch, you'd expect to see the COM macros to implement IDispatch.

However, no matter how long you comb through the source code for CCmdTarget, you'll see no mention of the MFC macros for implementing COM classes. Hmm. What's happening? How does CCmdTarget implement IDispatch?

There's a file in the MFC source code directory called OLEIMPL2.H that defines a class called COleDispatchImpl. This class contains the seven functions for IDispatch. There are the three IUnknown functions—AddRef(), Release(), QueryInterface()—and the four functions defined by IDispatch: GetTypeInfoCount(), GetTypeInfo(), GetIDsOfName(), and Invoke(). So COleDispatchImpl defines a C++ class that implements IDispatch. It looks as if this is where MFC implements IDispatch. But if CCmdTarget implements IDispatch, how does COleDispatchImpl get hooked up to CCmdTarget? The answer lies in the function CCmdTarget::EnableAutomation().

CCmdTarget::EnableAutomation()

MFC's documentation states that you need to call CCmdTarget's EnableAutomation() function to get Automation rolling. EnableAutomation() is one of those black magic things you have to do in MFC-land that the documentation never explains (the documentation says, "Don't say nothing—just do it!"). EnableAutomation() connects the CCmdTarget to the COleDispatchImpl class.

If you go back and take a look at CCmdTarget's definition, you'll notice that CCmdTarget declares itself as a friend to COleDispatchImpl. (Remember, COleDispatchImpl is the class that provides the standard implementation of IDispatch.) CCmdTarget also declares a nested class called XDispatch, which has a single DWORD representing the IDispatch vtable:

```
struct XDispatch
{
    DWORD m_vtbl; // place-holder for IDispatch vtable
} m_xDispatch;
```

The goal here is to somehow get a regular vtable that implements IDispatch and attach it to CCmdTarget. That's just what EnableAutomation() does: it copies the address of COleDispatchImpl's vtable to a DWORD representing the IDispatch vtable that CCmdTarget needs. In normal mode (that is, using CCmdTarget without calling EnableAutomation()), CCmdTarget doesn't support IDispatch, and XDispatch::m_vtbl is initialized to zero. Calling EnableAutomation() sets XDispatch::m_vtbl equal to COleDispatchImpl's vtable. CCmdTarget always has an IDispatch vtable. It remains

NULL until you call `EnableAutomation()`. `CCmdTarget::EnableAutomation()` effectively plugs an `IDispatch` vtable into `CCmdTarget`.

So, at this point `CCmdTarget` has an `IDispatch` vtable. Where are the interface maps? It's not obvious how `CCmdTarget`'s `IDispatch` gets picked up during a `QueryInterface()`. Because MFC's Automation implementation works, there must be an entry for `IID_IDispatch` in `CCmdTarget`'s interface map. Indeed, when you step through the debugger watching MFC doing the interface map lookups using `InternalQueryInterface()`, you'll notice that the GUID for `IID_IDispatch` is always the first entry into `CCmdTarget`'s interface map. But how does it get there? Look inside the file `CMDTARG.CPP`. MFC declares the first entry into `CCmdTarget`'s interface map to have an IID of `IID_IDispatch`, with the vtable offset to be `CCmdTarget`'s `m_xDispatch` member.

```
const AFX_INTERFACEMAP_ENTRY CCmdTarget::_interfaceEntries[] =
{
    INTERFACE_PART(CCmdTarget, _afx_IID_IDispatch, Dispatch)
    { NULL, (size_t)-1 }      // end of entries
};
```

This way, whenever a client calls `QueryInterface()` on a `CCmdTarget`-derived class, the class always has an entry for `IID_Dispatch`. The value of the vtable pointer is either `NULL` or it has the value of `COleDispatchImpl`'s vtable, depending on whether or not `EnableAutomation()` was called.

Other Helpful `CCmdTarget` Functions

Besides implementing `IDispatch`, `CCmdTarget` has a couple of helpful member functions: `GetIDispatch()` and `FromIDispatch()`. `GetIDispatch()` retrieves the raw `IDispatch` interface pointer from `CCmdTarget`. This is handy if you ever need to use the live `IDispatch` pointer. `FromIDispatch()` returns the `CCmdTarget` class associated with a particular `IDispatch` pointer.

In addition to the `CCmdTarget` extensions, MFC uses dispatch maps to track down and access dispatch members according to `DISPIDs`. Here's how they work.

Dispatch Maps

The second component in MFC's version of `IDispatch` is a technology called *dispatch maps*. MFC has maps all over the place; why should Automation be any different? MFC uses dispatch maps to implement `IDispatch::GetIDsOfNames()` and `IDispatch::Invoke()`. Recall that the `DISPMATH` object simply switched on the `DISPIDs` to implement `Invoke()`. By contrast, MFC's dispatch maps provide a table-driven alternative to switch statements.

MFC uses macros (what a surprise!) to implement dispatch maps. MFC's dispatch map macros follow the same basic paradigm as all the other MFC maps. One macro introduces the member variables and functions necessary for dispatch maps. Naturally, this macro is called `DECLARE_DISPATCH_MAP`. Then there are two macros for implementing the actual member variables and functions. They are `BEGIN_DISPATCH_MAP` and `END_DISPATCH_MAP`. Finally, there's a set of macros used to fill the dispatch maps. MFC has four different macros for populating the dispatch map. There is one macro for defining Automation functions and three macros for defining Automation properties: `DISP_FUNCTION`, `DISP_PROPERTY`, `DISP_PROPERTY_NOTIFY`, and `DISP_PROPERTY_EX`.

If you generate your program using AppWizard, turning on the “Automation—yes, please” button yields an Automation-enabled document. That is, the document's constructor calls `EnableAutomation()` and the document gets a blank dispatch map that's ready to be filled in. Once a `CCmdTarget`-derived class has a dispatch map, you can use ClassWizard to add and remove properties and methods. Here's a look at the insides of the MFC dispatch map.

Declaring the Dispatch Map

By using `DECLARE_DISPATCH_MAP` in your header file, you set up your class with the necessary machinery to implement an `IDispatch` lookup. Here's what `DECLARE_DISPATCH_MAP` looks like:

```
#define DECLARE_DISPATCH_MAP() \
private: \
    static const AFX_DISPMAP_ENTRY _dispatchEntries[]; \
    static UINT _dispatchEntryCount; \
protected: \
    static AFX_DATA const AFX_DISPMAP dispatchMap; \
    virtual const AFX_DISPMAP* GetDispatchMap() const; \
```

`DECLARE_DISPATCH_MAP` declares a static array of `AFX_DISPMAP_ENTRY` structures called `_dispatchEntries`. It also sets up a variable to keep track of the size of the `IDispatch` lookup table. Then `DECLARE_DISPATCH_MAP()` adds an `AFX_DISPMAP` structure and a virtual function for retrieving the dispatch map, called `GetDispatchMap()`.

Here's the definition of the `AFX_DISPMAP` structure:

```
struct AFX_DISPMAP \
    const AFX_DISPMAP* pBaseMap; \
    \
    const AFX_DISPMAP_ENTRY* lpEntries; \
    UINT* lpEntryCount; \
};
```

AFX_DISPMAP contains a pointer to the base dispatch map, a pointer to the first element in the current class's dispatch map, and the number of entries in the dispatch map.

As you might expect, the dispatch map is filled with various dispatch methods and properties. These are represented by the AFX_DISPMAP_ENTRY structure:

```
struct AFX_DISPMAP_ENTRY
{
    LPCTSTR lpszName;           // member/property name
    long lDispID;               // DISPID (may be DISPID_UNKNOWN)
    LPCSTR lpszParams;          // member parameter description
    WORD vt;                   // return value type / or type of property
    AFX_PMSG pfn;               // normal member On<membercall> or, OnGet<proper
    AFX_PMSG pfnSet;             // special member for OnSet<property>
    size_t nPropOffset;          // property offset
    AFX_DISPMAP_FLAGS flags;     // flags (e.g. stock/custom)
};
```

We'll see the five macros MFC provides to fill in the dispatch map entries shortly. However, let's first examine BEGIN_DISPATCH_MAP and END_DISPATCH_MAP.

Implementing the Dispatch Map

In addition to declaring the dispatch map, your CCmdTarget-derived class has to do a little more setup using BEGIN_DISPATCH_MAP/END_DISPATCH_MAP. Let's start with BEGIN_DISPATCH_MAP.

Like the BEGIN... macros in the other various MFC lookup mechanisms (such as message maps and interface maps), BEGIN_DISPATCH_MAP starts the ball rolling by implementing GetDispatchMap(), filling in the dispatchMap structure, and assigning entries into the dispatch map table. Here's the actual definition of BEGIN_DISPATCH_MAP:

```
#define BEGIN_DISPATCH_MAP(theClass, baseClass) \
    const AFX_DISPMAP* theClass::GetDispatchMap() const \
    { return &theClass::dispatchMap; } \
    const AFX_DISPMAP theClass::dispatchMap = \
        { &baseClass::dispatchMap, &theClass::_dispatchEntries[0], \
          &theClass::_dispatchEntryCount }; \
    UINT theClass::_dispatchEntryCount = (UINT)-1; \
    const AFX_DISPMAP_ENTRY theClass::_dispatchEntries[] = \
    { }
```

Dispatch maps are terminated using the END_DISPATCH_MAP macro. END_DISPATCH_MAP adds a NULL entry to the end of the dispatch map lookup table and inserts the closing brace and semicolon:

```
#define END_DISPATCH_MAP() \
{ VTS_NONE, DISPID_UNKNOWN, VTS_NONE, VT_VOID, \
  (AFX_PMSG)NULL, (AFX_PMSG)NULL, (size_t)-1, afxDispCustom } }; \
```

Of course, the dispatch map is useless without any data in it. MFC has five macros that fill the dispatch map with Automation method and property lookup information. These macros are (1) DISP_FUNCTION, (2) DISP_PROPERTY, (3) DISP_PROPERTY_NOTIFY, (4) DISP_PROPERTY_EX, and (5) DISP_PROPERTY_PARM.

DISP_FUNCTION

When you use ClassWizard to insert an Automation method, ClassWizard places this macro into the dispatch map for your class. DISP_FUNCTION adds an Automation method in the dispatch map by filling the AFX_DISPMAP_ENTRY with the necessary information for the method:

```
#define DISP_FUNCTION(theClass, szExternalName, pfnMember, vtRetVal,
vtsParams) \
{ _T(szExternalName), DISPID_UNKNOWN, vtsParams, vtRetVal, \
  (AFX_PMSG)(void (theClass::*)(void))pfnMember, (AFX_PMSG)0, 0, \
  afxDispCustom }, \
```

DISP_FUNCTION takes five parameters: the name of the class for which this dispatch map is created, the name of the method as seen by the outside world (that is, the automation client), the C++ member function, a VARIANT tag representing the function's return value, and a list of VARIANT tags representing the parameters to the function DISP_FUNCTION.

This macro fills the AFX_DISPMAP_ENTRY structure with the name of the dispatch method, the value of DISPID_UNKNOWN, a description of the parameter, a VARIANT tag representing the type of the result, and a pointer to the C++ class's member function. Using ClassWizard to add an Automation method, ClassWizard adds a member function to your C++ class and uses the DISP_FUNCTION macro to create the dispatch map entry.

DISP_PROPERTY

If you use ClassWizard to add an Automation property without get/set access functions and without a notification function (a plain-vanilla property), ClassWizard uses the DISP_PROPERTY to insert an Automation property into the dispatch map.

```
#define DISP_PROPERTY(theClass, szExternalName, memberName, vtPropType) \
{ _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, (AFX_PMSG)0, \
  (AFX_PMSG)0, offsetof(theClass, memberName), afxDispCustom }, \
```

DISP_PROPERTY takes four parameters: (1) the name of the class to which this dispatch map is associated, (2) the external name of the property, (3) the name of the CCmdTarget-derived class's member variable, and (4) a VARIANT tag representing the type of the property.

This macro fills the AFX_DISPMAP_ENTRY with the name of the dispatch property, the value of DISPID_UNKNOWN, a NULL string (representing no parameters), a VARIANT tag representing the type of property, and the address of the C++ class's member variable. ClassWizard adds the member variable to the C++ class and attaches this macro to the dispatch map.

DISP_PROPERTY_NOTIFY

Sometimes you'd like to be able to know when an automation client has changed a property. By using ClassWizard and supplying the name of a notification function with your property, ClassWizard uses the **DISP_PROPERTY_NOTIFY** to insert an Automation property into the dispatch map as well as provide a notification function that is called whenever the property is changed by the Automation client.

```
#define DISP_PROPERTY_NOTIFY(theClass, szExternalName, memberName,
pfnAfterSet, vtPropType) \
{ _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, (AFX_PMSG)0, \
(AFX_PMSG)(void (theClass::*)(void))pfnAfterSet, \
offsetof(theClass, memberName), afxDispCustom }, \
```

This macro is more or less the same as the **DISP_PROPERTY** function except that it adds a pointer to a notification function. It assigns this function pointer to AFX_DISPMAP_ENTRY's pfnSet field. ClassWizard will add both the member variable and the notification function to your class.

DISP_PROPERTY_EX

In addition to adding a notification function for your properties, you can also expose your properties through a pair of Get and Set functions. **DISP_PROPERTY_EX** inserts an Automation property in the dispatch map and provides a pair of access functions.

```
#define DISP_PROPERTY_EX(theClass, szExternalName, pfnGet, pfnSet,
vtPropType) \
{ _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, \
(AFX_PMSG)(void (theClass::*)(void))pfnGet, \
(AFX_PMSG)(void (theClass::*)(void))pfnSet, 0, afxDispCustom }, \
```

DISP_PROPERTY_EX is very similar to **DISP_PROPERTY_NOTIFY**, except instead of providing a single notification function, this macro uses a pair of Get and

Set functions. Using ClassWizard to add a Get/Set Automation property yields Get...() and Set...() functions for the C++ class. This macro assigns the Get...() and Set...() functions to the the pfnGet and pfnSet fields of AFX_DISPATCH_ENTRY.

DISP_PROPERTY_PARAM

Sometimes you'd like to have Get...() and Set...() functions for particular functions that have extra parameters. For this, MFC provides the DISP_PROPERTY_PARAM macro. This macro is just like DISP_PROPERTY_EX—it defines an Automation property with Get/Set access functions. However, the Get/Set parameters here can have extra parameters.

```
#define DISP_PROPERTY_PARAM(theClass, szExternalName, pfnGet, pfnSet,
vPropType, vtsParams) \
{ _T(szExternalName), DISPID_UNKNOWN, vtsParams, vPropType, \
(AFX_PMSG)(void (theClass::*)(void))pfnGet, \
(AFX_PMSG)(void (theClass::*)(void))pfnSet, 0 }, \
```

Using ClassWizard to add this kind of property adds a pair of Get/Set functions to your C++ classes, whose signatures reflect the extra parameters.

Expanding the Macros

Here's an example of how the dispatch map macros work. This dispatch map is taken from a simple program called AUTOTEST. It's a regular MFC program whose document (CAutotestDoc) supports Automation. The dispatch map has one of each type of entry: a dispatch function, a dispatch property without notification, a dispatch property with notification, a dispatch property with Get/Set access functions, and a dispatch parameter whose Get/Set access functions have extra parameters.

Listing 14-5 shows the code left in ClassWizard's wake:

Listing 14-5. Dispatch map created by ClassWizard

```
BEGIN_DISPATCH_MAP(CAutotestDoc, CDocument)
//{{AFX_DISPATCH_MAP(CAutotestDoc)
DISP_FUNCTION(CAutotestDoc, "SomeMethod", SomeMethod, VT_I2,
VTS_I4 VTS_I2)
DISP_PROPERTY(CAutotestDoc, "PlainProperty", m_plainProperty, VT_I4)
DISP_PROPERTY_NOTIFY(CAutotestDoc, "NotifyProperty",
m_notifyProperty, OnNotifyPropertyChanged, VT_R8)
DISP_PROPERTY_EX(CAutotestDoc, "Get SetProperty",
GetGetProperty, SetGetProperty, VT_R4)
DISP_PROPERTY_PARAM(CAutotestDoc, "Get SetProperty2",
GetGetProperty2, SetGetProperty2, VT_I4, VTS_I2 VTS_I4)
//}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

When you compile the code, you get a full-fledged dispatch map as shown in Listing 14-6. The spaces between each entry are for clarity.

Listing 14-6. Dispatch maps expanded by the preprocessor

```
const AFX_DISPMAP* CAutotestDoc::GetDispatchMap() const {
    return &CAutotestDoc::dispatchMap; }

const AFX_DISPMAP CAutotestDoc::dispatchMap = {
    &CDocument::dispatchMap,
    &CAutotestDoc::_dispatchEntries[0],
    &CAutotestDoc::_dispatchEntryCount
};

UINT CAutotestDoc::_dispatchEntryCount = (UINT)-1;
const AFX_DISPMAP_ENTRY CAutotestDoc::_dispatchEntries[] = {
    { "SomeMethod", ( -1 ), "\x03" "\x02", VT_I2,
        (AFX_PMSG)(void (CAutotestDoc::*)(void))SomeMethod, (AFX_PMSG)0, 0 },

    { "PlainProperty", ( -1 ), 0, VT_I4,
        (AFX_PMSG)0, (AFX_PMSG)0,
        (size_t)&(((CAutotestDoc *)0)->m_plainProperty) },

    { "NotifyProperty", ( -1 ), 0, VT_R8,
        (AFX_PMSG)0, (AFX_PMSG)
        (void (CAutotestDoc::*)(void))OnNotifyPropertyChanged,
        (size_t)&(((CAutotestDoc *)0)->m_notifyProperty) },

    { "Get SetProperty", ( -1 ), 0, VT_R4,
        (AFX_PMSG)(void (CAutotestDoc::*)(void))GetGetProperty,
        (AFX_PMSG)(void (CAutotestDoc::*)(void))SetGetProperty, 0 },

    { "Get SetProperty2", ( -1 ), "\x02" "\x03", VT_I4,
        (AFX_PMSG)(void (CAutotestDoc::*)(void))GetGetProperty2,
        (AFX_PMSG)(void (CAutotestDoc::*)(void))SetGetProperty2, 0 },

    { 0, ( -1 ), 0, VT_VOID, (AFX_PMSG)0, (AFX_PMSG)0, (size_t)-1 }
};
```

MFC and DISPIDs

Examine the dispatch map closely. What? All the entries have a dispatch ID of -1 (which corresponds to DISPID_UNKNOWN). What's going on here? As it turns out, the DISPIDs aren't hard-coded: MFC figures out the dispatch IDs on the fly. Hmm... Let's see what happens when an Automation client tries to get the dispatch ID of a member function or property.

CCmdTarget and GetIDsOfNames()

GetIDsOfNames() is the function that looks up a dispatch property's DISPID given its human-readable name. Under MFC, GetIDsOfNames() is implemented by the COleDispatchImpl helper class. GetIDsOfNames() takes five parameters, the most important of which are the array of strings representing the dispatch member names and an empty array of dispatch IDs:

```
HRESULT GetIDsOfNames(REFIID riid,
                      LPTSTR* rgszNames,
                      UINT cNames,
                      LCID,
                      DISPID* rgdispid)
```

GetIDsOfNames() without Type Information

As with all its maps, MFC maintains dispatch maps for a class hierarchy in a list. Each Automation-enabled class in the hierarchy has its own dispatch map. Each dispatch map is an array of AFX_DISPMAP_ENTRY structures. To illustrate this, consider two Automation-enabled classes (derived from CCmdTarget, of course) called CMath and CMathEx. CMath exposes two virtual methods: Add() and Subtract(). CMathEx implements Multiply() and Divide() in addition to inheriting Add() and Subtract(). CMath and CMathEx each have their own dispatch maps. Then each of the dispatch maps is linked backward, starting with the most-derived class and ending with the base class (CCmdTarget). Figure 14-3 illustrates this relationship.

Note that if you had a CMath object all by itself then the DISPIIDs for Add and Subtract would change since there would be no base class involved. This surprises some people when they realize that MFC's implementation doesn't allow for polymorphic access to CMath and CMathEx as far as their DISPIIDs are concerned. However, this feature of "object orientedness" isn't really a feature of automation. You can use inheritance for implementing your object, but as far as the client is concerned two different kinds of objects are exactly that—unrelated implementations.

MFC uses this arrangement to create DISPIIDs on the fly. MFC divides dispatch IDs (which are DWORDS) into two parts. The high word represents the distance of the dispatch member from the current class (that is, the most-derived class). The low word represents the index of the entry into the dispatch map. So in the preceding example, the DISPID for CMath's Add() function is 00010001, and the dispatch ID for CMath's Subtract() function is 00010002. The high word is 1 because Add() and Subtract() belong to CMath, which is one class removed from CMathEx. The dispatch ID for CMathEx's Multiply() function is 00000001, and the dispatch ID for CMathEx's Divide() function is 00000002. The high word is 0 because Multiply() and Divide() belong to CMathEx, the most-derived class (that is, the distance from the most-derived class is zero). The low word represents the position in CMath's dispatch map.

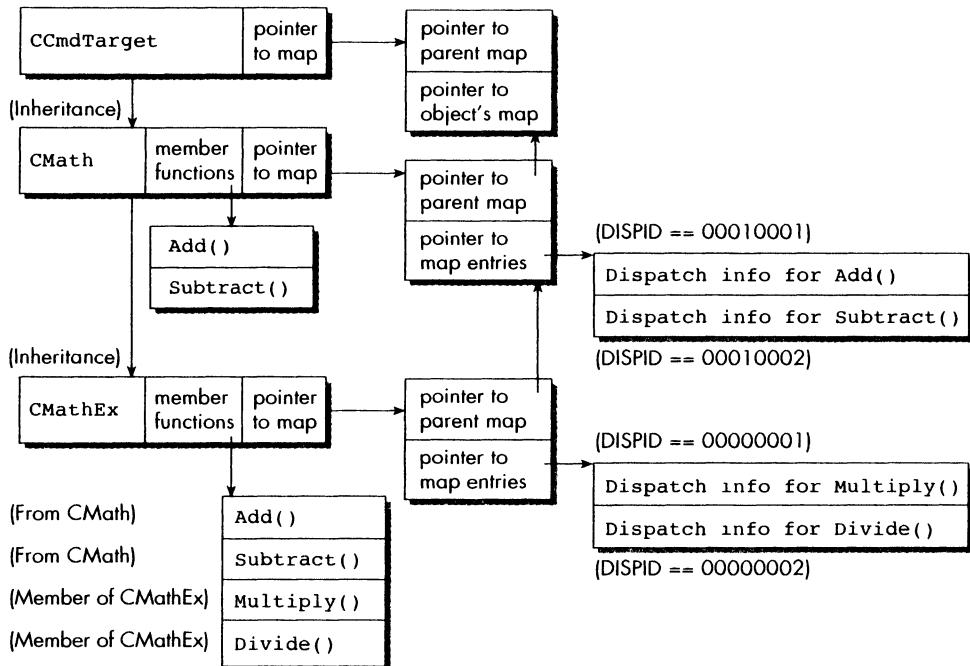


Figure 14-3. A pictorial representation of a dispatch map

MFC generates the DISPID on the fly by looking through the class's dispatch maps. To generate the DISPID, the CCmdTarget-derived class first gets a pointer to its dispatch map using the GetDispatchMap() function (which was provided by the macros). Then GetIDsOfNames() calls CCmdTarget::MemberIDFromName(), passing the object's dispatch map and a pointer to the first element in an array of dispatch IDs provided by the caller trying to figure out the DISPID from the human-readable name.

Remember that each entry in the dispatch map contains a string identifying the dispatch member. CCmdTarget::MemberIDFromName() starts at the most-derived class's dispatch map searching for a name that matches one of the dispatch members. If it can't find a match in the most-derived class's dispatch map, it moves to the next dispatch map (on the way toward CCmdTarget). As MemberIDFromName() traverses the list of dispatch maps, it keeps track of its position in the hierarchy. When it finds a match, it melds the index of the dispatch map with the index of the dispatch member to form a dispatch ID. GetIDsOfNames() returns this value to the Automation client so that the Automation client can use it to invoke methods and access properties.

This seems like a funky way of generating DISPIIDs. However, you'll soon see that Microsoft did this to optimize Invoke().

CCmdTarget and Invoke()

Again, CCmdTarget doesn't actually implement IDispatch::Invoke(). CCmdTarget uses the helper class COleDispatchImpl. COleDispatchImpl::Invoke() has the standard Invoke() signature:

```
STDMETHODIMP ColeDispatchImpl::Invoke(
    DISPID dispid, REFIID riid, LCID lcid,
    WORD wFlags, DISPPARAMS* pDispParams,
    LPVARIANT pvarResult,
    LPEXCEPINFO pexcepinfo, UINT* puArgErr);
```

COleDispatchImpl::Invoke() calls CCmdTarget::GetDispEntry() to retrieve the dispatch map entry. GetDispEntry() uses the high word of the DISPID to cut to the chase and find the correct dispatch map. Then GetDispEntry() uses the low word of the DISPID to jump straight to the correct dispatch map entry. This lookup turns out to be very fast because the code doesn't have to traverse the dispatch maps (remember, that was already done in GetIDsOfNames()).

Most of the work done by Invoke() involves error checking—making sure that the number of parameters and the result types are in order. If the dispatch member is a method or a Get/Set property, then Invoke() calls CCmdTarget::CallMemberFunction() to invoke the function or access the property. If the dispatch member is a regular property (or a property with a notification function), Invoke() calls CCmdTarget::GetStandardProperty() or CCmdTarget::SetStandardProperty(), depending on whether the client is getting or setting the property.

CallMemberFunction()

CallMemberFunction() pulls apart the parameter list and sets up the call stack so that the class's member function can deal with the arguments. Once the stack frame is built, CallMemberFunction() uses _AfxDispatchCall() to set up the real stack. _AfxDispatchCall() does some assembly language hocus-pocus to point the real runtime stack to the stack fabricated by CallMemberFunction(). Then _AfxDispatchCall() calls the function specified by the dispatch map entry's pfn member. Notice that if the dispatch member is a Get/Set property, the dispatch map entry's pfn field points to the appropriate Get...() or Set...() function.

SetStandardProperty() and GetStandardProperty()

If the dispatch member is a property, CCmdTarget has two functions to get and set plain-vanilla properties: CCmdTarget::SetStandardProperty() and CCmdTarget::GetStandardProperty().

When you add a regular dispatch property or a dispatch property with notification, ClassWizard adds a member variable to your C++ class. At the same time, ClassWizard adds an AFX_DISPMAP_ENTRY to the class's dispatch map. The last field in the AFX_DISPMAP_ENTRY structure is the offset of the member variable, called nPropOffset. MFC needs this address to access the variable whenever the client wants to set the property. Inside SetStandardProperty(), MFC fills the memory location occupied by the member variable with the value specified by the client. If the property is a notification property, the pfnSet field of the AFX_DISPMAP_ENTRY points to the class's notification function. That way, whenever the client changes the dispatch property, SetStandardProperty() calls the notification function.

GetStandardProperty() works the same way, but in the opposite direction. That is, GetStandardProperty() takes the value located in the position represented by the nPropOffset field in the AFX_DISPMAP_ENTRY structure and stuffs it into the VARIANT representing the result. Invoke() eventually passes this value back to the Automation client.

As you can see, MFC covers basic Automation fairly well. But what about this type information thing? How do Visual C++ and MFC deal with this increasingly important aspect of OLE?

MFC and Type Information

An Automation server isn't really complete until you provide a type library for it. As described earlier, type libraries contain information about a COM class. Type libraries are becoming pivotal to the success of Automation—especially when it comes to OLE controls. OLE control containers depend heavily upon type information to implement OLE control event technology. In addition, developers using Visual C++ can use a type library to very quickly write a C++ client for your Automation object. Fortunately, the AppWizards for Visual C++ versions 2.0 and greater generate an ODL file that can be compiled into a type library. ClassWizard even keeps the ODL file up to date as you add properties and methods.

Until OLE controls came along, that represented Visual C++'s and MFC's entire support for type libraries. For a long time, MFC didn't support the two IDispatch member functions that deal with type information: GetTypeInfo() and GetTypeInfoCount(). MFC's earlier implementation of IDispatch (in the COleDispatchImpl class), GetTypeInfo(), and GetTypeInfoCount() simply returned the error code of E_NOTIMPL.

Now with the advent of MFC version 4.0, better support for type information is emerging. We'll examine the type library and type info support briefly for now. We'll cover it in more detail when we get to OLE controls, in Chapter 15, because that's where type libraries are used the most.

MFC has two macros, **DECLARE_OLETYPELIB** and **IMPLEMENT_OLETYPELIB**, for adding type library support and a number of CCmdTarget member functions for dealing with type information.

Type Library Macros

Like MFC's support for class factories, MFC has macros for supporting type libraries. MFC has a **DECLARE_OLETYPELIB** and an **IMPLEMENT_OLETYPELIB** macro.

To add type library support to your Automation object, place the **DECLARE_OLETYPELIB** macro in your object's header file. The macro looks like this:

```
#define DECLARE_OLETYPELIB(class_name) \
protected: \
    virtual UINT GetTypeInfoCount(); \
    virtual HRESULT GetTypeLib(LCID, LPTYPELIB*); \
    virtual CTypeLibCache* GetTypeLibCache(); \
```

DECLARE_OLETYPELIB adds three virtual functions to your Automation class: **GetTypeInfoCount()**, **GetTypeLib()**, and **GetTypeLibCache()**. As you might expect, **IMPLEMENT_OLETYPELIB** finishes the job by implementing the virtual functions:

```
#define IMPLEMENT_OLETYPELIB(class_name, tlid, wVerMajor, wVerMinor) \
UINT class_name::GetTypeInfoCount() \
{ return 1; } \
HRESULT class_name::GetTypeLib(LCID lcid, LPTYPELIB* ppTypeLib) \
{ return ::LoadRegTypeLib(tlid, wVerMajor, wVerMinor, lcid, \
ppTypeLib); } \
CTypeLibCache* class_name::GetTypeLibCache() \
{ return AfxGetTypeLibCache(&tlid); } \
```

IMPLEMENT_OLETYPELIB takes four parameters: (1) the class for which the type library is meant to contain information, (2) the type library's GUID, and (3) the major and (4) minor versions of the library. **IMPLEMENT_OLETYPELIB** proceeds to implement the functions declared by **DECLARE_OLETYPELIB**. **GetTypeInfo()**'s a no-brainer: it returns 1, meaning that the object has type information. **GetTypeLib()** is just a wrapper for the API function **LoadRegTypeLib()**. Finally, **GetTypeLibCache()** returns a pointer to the module's type library cache. But where does the type library cache come from?

The Type Library Cache

MFC maintains a cache of type library pointers for Automation applications that support type information. You'll find this map in the AFX_OLE_STATE structure. This structure includes a CMapPtrToPtr member called m_mapExtraData:

```
CMapPtrToPtr m_mapExtraData;
```

The name m_mapExtraData will probably change in the future. Microsoft is constantly tuning the performance of MFC. In doing so, they have changed the implementation to be optimized for one type library per module (the most common case). When MFC sees more than one type library cached, MFC goes to the map. It could be that when Microsoft originally introduced this map (way back in 16-bit CDK) it was used for more than the typelib cache.

This m_mapExtraData maintains a dictionary of CTypeLibCache structures. The dictionary maps a type library's GUID to an instance of CTypeLibCache. CTypeLibCache wraps an OLE type library, providing functions for extracting type information out of the type library. The definition appears in Listing 14-7.

Listing 14-7. MFC's CTypeLibCache class

```
class CTypeLibCache
{
public:
    CTypeLibCache(const GUID* ptlid);
    ~CTypeLibCache();

    void Lock();
    virtual void Unlock();
    BOOL Lookup(LCID lcid, LPTYPELIB* pptlib);
    void Cache(LCID lcid, LPTYPELIB ptlib);
    BOOL LookupTypeInfo(LCID lcid, REFGUID guid, LPTYPEINFO* pptinfo);
    void CacheTypeInfo(LCID lcid, REFGUID guid, LPTYPEINFO ptinfo);

protected:
    const GUID* m_ptlid;
    LCID m_lcid;
    LPTYPELIB m_ptlib;
    GUID m_guidInfo;
    LPTYPEINFO m_ptinfo;
    long m_cRef;
};
```

MFC has a global function called AfxGetTypeLibCache(). AfxGetTypeLibCache() takes a single parameter: a GUID representing the type library. AfxGetTypeLibCache() uses this GUID to look up the matching CTypeLibCache class.

CCmdTarget and the CTypeLibCache work in concert to provide type library support.

CCmdTarget Support for Type Libraries

CCmdTarget has six member functions for supporting type libraries:

1. CCmdTarget::EnableTypeLib() gets the application's type library cache and places a reference count on it (using CTypeLibCache::Lock()).
2. CCmdTarget::GetTypeInfoOfGuid() retrieves specific type information from a CCmdTarget's type library.
3. CCmdTarget::GetDispatchIID() is a virtual function that needs to be overridden in the derived class. CCmdTarget's IDispatch implementation uses this function when calling IDispatch::GetTypeInfo(). GetDispatchIID returns the GUID representing type information for the CCmdTarget.
4. CCmdTarget::GetTypeInfoCount() is virtual as well. It gets overridden automatically by the type library macros.
5. CCmdTarget::GetTypeLibCache() is also overridden by the type library macros. This is also a virtual function.
6. CCmdTarget::GetTypeLib(), also overridden by the macros, is a wrapper for the OLE API function LoadRegTypeLib().

Type library support is explicitly provided by Visual C++'s ControlWizard. That is, whenever you create an OLE control, ControlWizard adds the macros and overrides GetDispatchIID for you. Until AppWizard supplies this level of support for regular Automation objects, you'll need to add this support by hand.

Conclusion: The Consequences of "The MFC Way"

All in all, MFC has a pretty decent IDispatch implementation. MFC probably uses this dispatch map technology because it fits right in with other, similar technologies like message maps. This makes it possible for Microsoft to easily integrate ClassWizard into the tool set. ClassWizard already has to deal with message maps. Adding Automation support to ClassWizard using dispatch maps is a reasonable extension.

MFC's implementation of IDispatch has good performance. The dispatch member lookup is flexible because you can use dispatch properties and methods of derived

classes very easily. However, in addition to this flexibility, MFC's version of IDispatch turns out to be zippy. The only performance hit is doing the actual dispatch ID lookup. Hopefully, clients are smart enough to call this function once per dispatch member and cache the result. Once the lookup is done, MFC knows exactly where to look in the dispatch map to find the necessary information to access the property or invoke the method.

Finally, one drawback of MFC's implementation of IDispatch is that it won't support dual interfaces very easily (at least for now). If you want to use dual interfaces, you've got to tweak the ODL file emitted by AppWizard (breaking ClassWizard in the process), use the Typing Wizard to add the COM interface support (you know—all that interface map stuff we covered in Chapter 11), and delegate to MFC for IDispatch implementation. This can be important if many of your clients are already using IDispatch; MFC's IDispatch is much faster than that supplied by the type info driven implementation.

OLE Controls

We saved talking about OLE controls until the end because they are at the top of the OLE food chain: they represent OLE's crowning jewel. OLE controls use the following OLE technologies:

- Automation technology
- In-place activation technology
- Persistence technology
- Connection technology
- Property page technology

Software developers have wanted the ability to develop applications in a “visual” way for a long time. Just as you’re able to pick up a regular control (like a push button) off the tool palette in the dialog box editor, wouldn’t it be great to be able to pick up a full-fledged software component that really does something and drop it into your application? For example, perhaps you’d like to easily add a component to drive a communications port. Well, OLE controls are the way to do it.

Component software is what OLE controls are all about! If you’ve done any of the control tutorials that come with Visual C++, then you’re undoubtedly familiar with how easy it is to create OLE controls. Like OLE documents, OLE controls require a lot of boilerplate code to get up and running. According to the Microsoft OLE control specification, OLE controls have to implement some 18 interfaces. Whoa! That’s a lot of functions.

However, once the boilerplate code is in place, writing an OLE control is a fairly painless process. As a framework for OLE controls, MFC fulfills the interface requirements for OLE controls. Just use ControlWizard to create a new control, add some painting code and some events, and you can have an OLE control that leads a

rich and fulfilled life. Of course, you can add other features like license validation and property notifications, as well.

However, as with all things OLE, the goings on under the hood are nontrivial. OLE controls are no different from other parts of OLE. As you can see from the list of technologies implemented by OLE controls, there's a lot involved. In this chapter, we cover the insides of MFC's OLE control implementation.

OLE controls could fill a book all by themselves. We don't have room to cover everything here, but we'll make sure to go over the most relevant topics. So hold on as we explore one of the most important OLE technologies available today, starting with why there are even such things as OLE controls.

VBXs and Their Shortcomings

In the early 1990s, Microsoft introduced a way to extend Visual Basic programs using something called Visual Basic Controls (a.k.a. VBXs). VBXs provided a path for developers to enhance Visual Basic programs using custom controls. In addition, these custom controls were not necessarily confined to Visual Basic programs: they could be applied to Windows applications developed in C++ or even Turbo Pascal for that matter. Eventually, this movement spawned a whole cottage industry. Companies began offering products such as spreadsheet components and communication components in the form of VBXs.

However, the VBX standard has several shortcomings. First, the VBX API arose as sort of an afterthought. It's not well thought out. As a result, there are three different versions of the VBX specification that work with various platforms. The second problem with VBXs is that the VBX API is inherently 16-bit. For example, the VBX API uses based pointers in many places. Finally, the biggest issue is that Microsoft has publicly announced that the VBX standard will not be carried forth into the 32-bit world. Instead, Microsoft offers an alternative: OLE controls.

OLE Controls

Enter OLE controls—the custom control standard of the future. As you can see from the previous list, OLE controls involve almost all the currently available OLE technologies. That means you can use an OLE control as an in-place object (as with compound documents), as an Automation object (because OLE controls implement IDispatch), or simply as a data source (because they implement IDataObject).

In addition to being in-place objects and supporting regular Automation (that is, an incoming Automation interface), OLE controls also support OLE Automation going the other way (that is, an outgoing Automation interface). In other words, OLE control containers implement IDispatch for the controls to call. OLE controls implement outgoing Automation as events. OLE controls implement events so that they can notify their clients when something interesting happens. For example, you may write a stock exchange monitor that monitors a live feed coming from the New York Stock Exchange. Using OLE control event technology, the OLE can notify its host of changes in certain stock prices. This sort of thing doesn't work in regular Automation. Regular Automation supports only incoming Automation interfaces. You could also notify the host of a change via an OLE control notification property. An OLE control notification property is a regular property (one that you might add using ClassWizard) that can send a notification to the host whenever the property changes.

As you can imagine, implementing all this boilerplate code isn't exactly a walk in the park. Clearly, a great deal goes into making an OLE control. But MFC makes it easy. Let's see what's involved in creating one.

Writing a Control

MFC makes writing an OLE control quite painless. Visual C++ includes a tutorial showing how to build a simple OLE control. In addition, there are many good articles available describing the process. Basically, writing an OLE control involves starting a new project as an OLE control. Doing this yields a simple OLE control that draws a circle in its presentation area.

Of course, once the control is compiled and linked, you need to register the control in the system registry. This isn't a big deal, because Microsoft has defined two new well-known functions for controls to expose: DllRegisterServer() and DllUnregisterServer(). DllRegisterServer() performs the necessary steps to register a control in the system registry. DllUnregisterServer() performs the opposite operation. Visual C++ includes a utility program called REGSVR32.EXE that loads an OLE control DLL and calls its DllRegisterServer() function to register the control in the registry. As its name implies, DllUnregisterServer() undoes the registration and cleans up the registration database. Visual C++ includes REGSVR32.EXE in its toolbar. To register your newly compiled server, just select Register Control from the Tools menu. This starts the REGSVR32 program and registers the control.

To make the control do something useful, you need to decide what type of user interface your control should have, as well as how the control should appear. In

addition, you need to decide what sorts of events the control should broadcast. For example, the stock market monitor in our example might broadcast such events as a change in the price of a certain stock.

In addition to broadcasting events, OLE controls also support regular plain-vanilla Automation. That way, OLE control clients can set properties and invoke functions on the control. In the stock monitor example, you might want to allow the control host to set the name of the company being monitored.

Of course, it doesn't make sense to have OLE controls without anyplace to put them. Let's see how you'd use an OLE control in an application.

Using OLE Controls in a Project

MFC supports using OLE controls in two contexts: within a dialog box or by themselves in a window. Visual C++'s support for OLE controls inside a dialog box is excellent. However, using controls within a non-dialog box window is a manual proposition. Though it's easy to plant a control within a non-dialog box window, you have to rig up the event-response mechanism yourself. Don't worry, though. You'll understand how to do that by the end of this chapter. Let's start by seeing how to put a control inside a dialog box.

OLE Controls in a Dialog Box

Adding an OLE control to a dialog box is quite easy using a new Visual C++ feature called the Component Gallery. The Component Gallery is a convenient mechanism for merging prewritten software components into your application. Naturally, that's where you'll find the OLE controls.

Pulling down the Component Gallery yields a dialog box with several tabbed pages. One of those pages is filled with icons representing the OLE controls listed in the registry. To add an OLE control to your project, just highlight the icon and press the Insert button. The Component Gallery generates a C++ class based on the OLE control's type information. Then, next time you use the dialog box editor, you'll see the OLE control's icon in the tool palette. To add the control to your dialog box, just choose the OLE control from the palette and place it on the dialog box. Figure 15-1 illustrates an example.

Though there's great ClassWizard support for using OLE controls in a dialog box, using an OLE control within a non-dialog box window requires a little more typing. However, it's not impossible. Here's how to add an OLE control to a view.



Figure 15-1. The stocks control inserted in a dialog box and the control icon in the palette

Using an OLE Control in a View

You need to perform several steps to use an OLE control in a view: (1) add the control to the project, (2) add an instance of the control to your view, and (3) create the control.

The first step is to get the control into your project. Use the Component Gallery to add the control to your project. Select Insert|Component from the main menu. Then choose the control you want to use from the OLE control tab and insert it into your project.

Once the control's inside your project, you need to declare an instance of the control inside your view. Just declare a member variable inside your view definition. For example, if you want to put the stocks control in your view, just declare the stocks control as a member variable.

Next, define a number for your control. When you create the control, it needs to be assigned a number (just as regular Windows controls have a control ID).

Placing a control in a view isn't difficult. Since the OLE control is represented by a CWnd on the container side, you can plant the control in the view as you would any other Windows control. The best place to create the control is within your view's WM_CREATE message handler. Create the control, passing the caption, the window style flag, the positioning rectangle, a pointer to the parent window, and the control ID:

```
m_stocks.Create(NULL, WS_VISIBLE,
                  CRect(150,150,250,250),
                  this, ID_STOCKSCTL);
```

Doing this much puts a control on your screen. It'll draw itself and respond to window messages, but the container is still turning a deaf ear to the control. In order for the container to receive events, the container has to implement an event sink. ClassWizard does this automatically for OLE controls in dialog boxes. However, to add event support to a non-dialog window, you need to type in some MFC event sink macros. We'll get to that when we cover events. (See the developer tip later in this chapter.)

This may all seem pretty magical. However, OLE controls are just implementing a high-level OLE protocol.

So How Does It All Work?

As you can see, writing an OLE control or an OLE control container isn't too terribly difficult if you use MFC and Visual C++. Adding Automation members (functions and methods) and events to an OLE control is almost trivial using the ClassWizard. Likewise, responding to OLE control events in a dialog box is equally easy. However, none of this stuff occurs through magic. MFC happens to implement a great deal of code to get all this working. First we'll examine MFC's OLE control support classes. Then we'll watch the interaction between an OLE control and an OLE control container all the way through from creation to termination of the control, covering such topics as (1) how MFC manages OLE control containers, (2) how an MFC control container creates the control, (3) how MFC implements OLE connections, and (4) how MFC handles OLE control events.

MFC's support for OLE controls is split between the OLE control and OLE control container classes. Here's an overview of those classes.

MFC's OLE Control Classes

Take another look at the list of things that an OLE control is capable of doing: in-place embedded object support, Automation, property pages, and connection and event technology. That's a pretty tall order to fill. Maybe that's why MFC sets aside some 25 source code files specifically for OLE controls!

MFC introduces several new classes specifically for implementing OLE controls. These classes include the following:

- COleControlModule
- COleControl
- COlePropertyPage
- COleConnectionPoint
- CPictureHolder
- CFontHolder
- CPropExchange

Here's a quick look at each of these classes.

COleControlModule

COleControlModule represents an OLE control's server. It's derived from CWinApp, so it's got all the functionality of CWinApp. Otherwise, this class is pretty lightweight. The control module does need a place to initialize and clean up after itself, so it uses CWinApp::InitInstance() and CWinApp::ExitInstance() for this purpose.

As a result, COleControlModule's definition is pretty simple:

```
class COleControlModule : public CWinApp
{
    DECLARE_DYNAMIC(COleControlModule)
public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
};
```

The real action for OLE controls happens in COleControl.

COleControl

Here's the main class for OLE controls. COleControl encapsulates a single control. This class handles the bulk of a control's functionality. It derives from CWnd, so it inherits all of CWnd's functionality. In addition, COleControl has support for handling events. Because COleControl is derived ultimately from CCmdTarget, COleControl handles IDispatch (so that OLE controls support Automation properties and methods).

Because COleControl is the main OLE controls class, you can imagine that it supports some OLE interfaces. Actually, COleControl implements quite a few interfaces. Look in CTLCORE.CPP—you'll see an interface map including all these interfaces:

- IOleObject—This is the main interface by which embedded in-place objects talk to their containers. It has the functions DoVerb() and Close().
- IConnectionPointContainer—This interface allows the client to ask the control whether or not it supports connection points.
- IOleControl—This interface extends the functionality of in-place objects by providing functions necessary for OLE controls.
- IPersist—This interface returns a COM class's GUID.
- IPersistMemory—This is a new interface designed specifically for controls (it's defined in OLECTL.H). It provides a mechanism for very fast persistence. IPersistMemory is used instead of IPersistStreamInit in order to get a bit more performance. Instead of dealing with persistence as a stream of data that you must read into a buffer before looking at it, IPersistMemory gives the control direct access to the container's memory buffer.

- **IPersistStreamInit**—Controls implement this interface so that control containers can tell a control to persist.
- **IOleInPlaceObject**—IOleInPlaceObject gives the container a way to activate and deactivate in-place objects, as well as manage how much of the in-place object should be visible.
- **IOleInPlaceActiveObject**—OLE document content objects implement IOleInPlaceActiveObject to provide a direct channel between (1) an in-place object, (2) the outermost frame window of the object's server application, and (3) the document window within the container application. The interface handles translating messages, managing the state of the frame window (activated or deactivated), and managing the state of the document window (activated or deactivated). The interface also informs the content object when it needs to resize its borders, and it manages modeless dialog boxes.
- **IDispatch**—Used to expose a control's Automation properties and methods.
- **IOleCache**—The IOleCache interface provides control of the presentation data that gets cached inside of an object. Cached presentation data is available to the container of the object even when the server application is not running or is unavailable.
- **IViewObject**—The IViewObject interface enables an object to display itself directly without passing a data object to the caller. In addition, this interface can create and manage a connection with an advise sink so that the caller can be notified of changes in the view object.
- **IViewObject2**—The IViewObject2 interface is an extension to the IViewObject interface, which returns the size of the drawing for a given view of an object. You can prevent the object from being run if it isn't already running by calling this method instead of IOleObject::GetExtent.
- **IDataObject**—This interface is used to notify containers of changes.
- **IPersistPropertyBag**—New for OLE controls, this interface is yet another way for a control to make its properties persist. Controls may implement this interface instead of IPersistPropertySet. IPersistPropertyBag is basically a fast way to do text-based persistence (such as the kind of persistent data you see in Visual Basic's .FRM files).
- **ISpecifyPropertyPages**—Used by the container to find out which property pages an OLE control supports.
- **IPerPropertyBrowsing**—This interface supports browsing properties one at a time (rather than a page at a time). It is used to implement user interfaces for

property modification similar to Visual Basic's property grid or Visual C++'s “All Properties” property page.

- **IProvideClassInfo**—This interface is implemented by the control and used by the container to retrieve type information for a control's events and notification properties.
- **IPersistStorage**—This interface is implemented by the control and used by the container to tell the control to persist into an **IStorage**.

Because **COleControl** is MFC's main class for supporting OLE controls, this is where we'll be spending most of the time in this chapter.

COlePropertyPage

OLE controls are Automation objects that expose properties via **IDispatch**. Most OLE controls allow the user to access these properties through property pages. These property pages are usually grouped together and shown as a tabbed dialog box. **COlePropertyPage** handles displaying OLE control properties in a dialog box.

COlePropertyPage derives from **CDialog**, so it has all the standard things that make using MFC dialogs so easy. For example, **COlePropertyPages** fully support DDX/DDV. In addition, **COlePropertyPages** also extend the data exchange mechanism to swap properties between the property page and the control itself.

COleConnectionPoint

One thing that distinguishes OLE controls from regular plain-vanilla Automation is that OLE controls can (1) expose events to their hosts and (2) notify their hosts when properties change. OLE controls accomplish this through connection technology. The **COleConnectionPoint** represents an OLE control's outgoing interfaces. Don't worry, we'll cover this in detail when we go over OLE control events.

CPictureHolder

CPictureHolder exists to implement a picture property for your control. Using the **CPictureHolder**, you can specify a bitmap, icon, or metafile for display and treat it uniformly.

CFontHolder

CFontHolder is similar to **CPictureHolder**. **CFontHolder** encapsulates the functionality of a Windows font object and the **IFont** interface. It's used to implement the stock font property. You can use this class to implement custom font properties for your own control.

CPropExchange

Given that OLE controls have properties, you probably want these properties to have the ability to persist. CPropExchange (and its derivatives) support property persistence for OLE controls.

On the other side of the fence, MFC includes some classes for implementing OLE control containers. Let's take a look at them before we dig into the real meat of OLE controls.

MFC and OLE Control Containers

Until late 1995, the only work you could do with OLE controls was to create them. Microsoft provided a handy OLE control test container, but it never shipped the source code—there was no easy way to see what was going on. MFC 4.0's OLE control container support finally completes the equation. While it has always been possible to roll your own control container from scratch, it isn't an attractive proposition and requires hand-coding several OLE interfaces. MFC and Visual C++ now make it very easy to use OLE controls inside your MFC-based applications—especially within dialog boxes. In fact, Microsoft has extended the resource editor and ClassWizard so that OLE controls appear like any other control. OLE controls now show themselves on the tool palette and control events appear in the message map tab in ClassWizard. In addition, OLE controls aren't confined to dialog boxes: you can plant them inside any CWnd-derived object!

MFC's support for OLE control containers lies largely inside three classes: COleControlContainer, COleControlSite, and COccManager.

COleControlContainer

COleControlContainer implements IOleInPlaceFrame and IOleContainer, which are necessary for OLE controls to fulfill their qualifications as in-place items. Within the scheme of OLE control containers, a CWnd-derived object that wants to contain controls maintains a COleControlContainer object as a member. In turn, the COleControlContainer class manages one or more COleControlSite objects.

COleControlSite

COleControlSite is mentioned once in the documentation as a parameter to CWnd::OnAmbientProperty(). Other than that, COleControlSite is undocumented. However, COleControlSite lies at the heart of MFC's OLE control container support.

COleControlSite represents the communication channel between the OLE control and the container application. There's one OLE control site for every OLE control inside a container application. In addition, the COleControlSite is also responsible for actually creating each single control in an OLE control container.

COleControlSite is a real COM object implementing several interfaces:

- IOleClientSite—Regular OLE document support.
- IOleInPlaceSite—Regular in-place activation support.
- IOleControlSite—This interface is implemented by the container and used by the control to notify the container of activation changes or to request activation information.
- IDispatch (for events)—This interface is implemented by the container and used by controls to inform the container of its events.
- IDispatch (for ambient properties)—The container implements this interface so the control can access the container's ambient properties.
- IPNotifySink—This interface, implemented by the container, is called by the control to notify the container that a property has changed, or to request an edit of a changing property.

We'll be seeing a lot more of COleControlSite as we follow the OLE control code to see how it works. Meanwhile, let's look at another class: COccManager.

COccManager

The COccManager class performs two services: (1) it manages OLE controls in a dialog box, and (2) it manages routing events to their proper destinations. MFC includes a global instance of this class, called `afxOccManager`.

That's a rundown of the OLE control container classes. Now it's time to watch the interaction between OLE controls and OLE control containers, as the container creates OLE controls, communicates through interfaces, and deals with events.

A Day in the Life of an OLE Control

Keep in mind that OLE controls are really just COM objects. They implement interfaces so that clients can communicate with them. Recall from Chapter 11 that (much of the time) COM objects materialize as a result of the client calling `CoCreateInstance()`. Remember also that `CoCreateInstance()` examines the registry to find the path to

the server implementing the COM object. Once `CoCreateInstance()` finds the server, it loads the server. In the case of an OLE control, the server is a DLL, so the Service Control Manager loads the DLL. Once the DLL is loaded, the operating system makes an entry into the well-known function `DllGetClassObject()`, retrieving the COM object's class factory and returning the `IClassFactory` interface pointer to the client. The client (still in the middle of calling `CoCreateInstance()` at this point) invokes `IClassFactory::CreateInstance()` to create an instance of the COM object, returning the requested interface.

At this point, the client uses the newly acquired interface to garner further interfaces and to begin an OLE-control-compliant dialog with the control. Let's watch this process up close—it's really quite interesting.

Control Creation

Controls have to come from somewhere. There are two ways to give birth to an OLE control in MFC: (1) add it to a dialog box template or (2) call the wrapper class's `Create()` function. Let's examine each case, starting with OLE controls in a dialog box.

Creating OLE Controls in a Dialog Box

Now that dialogs can include a new kind of control (OLE controls, that is), you can imagine that Microsoft had to modify their dialog templates to include new information specific to OLE controls. If you watch the OLE control creation code for dialog boxes, you'll see that the dialog template information has indeed changed.

OLE controls are created in the dialog box's `WM_INITDIALOG` handler. `CDialog::HandleInitDialog()` responds to the `WM_INITDIALOG` message. `HandleInitDialog()` retrieves the framework's `COccManager`, `afxOccManager`, then calls `afxOccManager`'s `CreateDlgControls()` using the dialog box template.

`COccManager::CreateDlgControls()` fetches the dialog template using `LoadResource()`. `COccManager::CreateDlgControls()` then cycles through each item (represented by a `DIALOGITEMTEMPLATE` structure) in the dialog box. For each OLE control in the dialog box, `COccManager::CreateDlgControls()` calls `COccManager::CreateDlgControl()` to actually create the control.

If the `DIALOGITEMTEMPLATE` represents an OLE control, then it includes GUIDs represented as strings. To create a control from the dialog templates, `CreateDlgControl()` transforms the class ID string into a GUID using the OLE API function `CLSIDFromString()`. `CreateDlgControl()` also gets the license key (if there's one needed to create the control).

The first parameter to `CreateDlgControl()` is the parent window (in this case, the dialog box). Remember that at the very outside, OLE controls live inside some

window (in this case, the window is the dialog box). Then the window maintains a COleControlContainer object, which in turn manages one or more COleControlSites. The CWnd class has a member variable named `m_pCtrlCont` of type COleControlContainer and a member function named `InitControlContainer()`, which creates a new COleControlContainer object and assigns it to CWnd::`m_pCtrlCont`. `CreateDlgControl()` calls `InitOleContainer()` to set up a COleControlContainer for the dialog box to hold controls.

COleControlContainer continues on the way to manufacturing controls by calling a member function named `CreateControl()`. As a result, COleControlContainer::`CreateControl()` tries to create a new COleControlSite, and in turn calls the COleControlSite's `CreateControl()` to create the actual control.

COleControlSite::`CreateControl()` initializes OLE if necessary, then calls COleControlSite::`CreateOrLoad()`. `CreateOrLoad()` uses an undocumented function, `CoCreateInstanceLic()` (found in OCCSITE.CPP), to create the control. Here's the prototype for `CoCreateInstanceLic()`:

```
HRESULT CoCreateInstanceLic(REFCLSID clsid, LPUNKNOWN pUnkOuter,
                           DWORD dwClsCtx, REFIID iid, LPVOID* ppv,
                           BSTR bstrLicKey)
```

Notice that it's exactly the same as the regular API function `CoCreateInstance()`, except it has a license key. If there's no license information, `CoCreateInstanceLic()` gets the control's class factory using `CoGetClassObject()`, requesting the IClassFactory interface. Then `CoCreateInstanceLic()` simply calls `IClassFactory::CreateInstance()` to create an instance of the control. If the control requires licensing, then `CoCreateInstanceLic()` gets the control's class factory using `CoGetClassObject()` but requests the IClassFactory2 interface (the second iteration of IClassFactory, where Microsoft decided this is the best place to handle component licensing). Then `CoCreateInstanceLic()` calls `IClassFactory2::CreateInstanceLic()`, passing the licensing key so the control can check against it to ensure that the control is being used by the host for which it was meant.

At this point, the thread of execution jumps to the control module. Remember from Chapter 11 that a call to a COM object's class factory's `CreateInstance()` results in the creation of the object. We'll see what happens on the other side of the fence soon. However, let's first see what happens when a stand-alone control is created.

Creating a Stand Alone Control

When you create OLE controls outside the context of a dialog box, you represent it using a regular CWnd-derived object. In our stocks example, you create a wrapper class for the stocks control by pulling it into your project using the Component Gallery.

The wrapper class derives from `CWnd` and includes member functions representing the control's Automation properties and methods. In addition, the wrapper class generated by the Component Gallery overrides `CWnd::Create()` (which is virtual). So if you want to add a stocks control to your view, just declare an instance of the wrapper class as a member to your view class. Then handle `WM_CREATE` by calling the wrapper class's `Create()` function like so:

```
m_stocks.Create(NULL, WS_VISIBLE,
                 CRect(150,150,250,250),
                 this, ID_STOCKSCTL);
```

The wrapper class's `Create()` function calls `CWnd::CreateControl()` to turn the `CWnd`-derived object into an OLE control site so it can host the control (remember, the actual control lives elsewhere in some DLL).

```
BOOL CWnd::CreateControl(REFCLSID clsid, LPCTSTR lpszWindowName,
                        DWORD dwStyle, const RECT& rect,
                        CWnd* pParentWnd, UINT nID, CFile* pPersist,
                        BOOL bStorage, BSTR bstrLicKey)
```

`CWnd::CreateControl()` calls the parent window's (the view's, in this case) `InitControlContainer`, which creates a `COleControlContainer` object member for the view and assigns it to the view's `m_pCtrlCont` variable. Then `CWnd::CreateControl()` calls the view's `m_pCtrlCont->CreateControl()`.

At this point, control creation is the same as for dialog boxes. The control site gets the control's class factory (using `CoGetObject()`) and then gets the class factory to generate an instance of the control. The control container asks for the control's `IOleObject` interface to be returned. The control container will need this interface to talk intelligently to the control.

Inside the Control

Once the DLL is in memory, the container makes a call into the control DLL's well-known exported function, `DllGetClassObject()`. `DllGetClassObject()` calls MFC's version: `AfxDllGetClassObject()`. Remember from Chapter 11 that `AfxDllGetClassObject()` walks the DLL's list of class factories trying to find the right class factory. If `AfxDllGetClassObject()` finds the correct class factory, it returns the `IClassFactory` pointer so that the control container can create the control. The control container uses the class factory's `CreateInstance()`, which ends up in the control's class factory. The control's class factory uses MFC's standard dynamic creation mechanism to create the control object.

Constructing the Control

Naturally, the thread of execution ends up in the control's constructor. MFC's default constructor for COleControl initializes all the member variables. It does the following:

- Calls EnableAggregation() to make the control aggregatable.
- Calls EnableAutomation() to turn on the IDispatch vtable for regular Automation.
- Calls EnableConnections() so the control and the control container can talk about connections (for example, events and property notifications). Incidentally, EnableConnections() works very much like EnableAutomation(). It plugs an IConnectionPointContainer vtable into the OLE control machinery.
- Calls AfxOleLockApp() to keep the DLL in memory for as long as the control is in use.

After COleControl::COleControl() is finished, the derived class's constructor is called. The ControlWizard inserts a call to InitializeIDs() in the control constructor. COleControl maintains GUIDs representing the control's primary dispatch interface and the events interface. InitializeIDs() initializes these member variables using the control's specific GUIDs. InitializeIDs() also calls EnableTypeLib() to turn on type library support for the control.

As an extra measure of safety, InitializeIDs() verifies that the type library contains the correct information. If the GUIDs in the CPP file and the ODL file do not match, MFC will trip some assertions here.

At this point, the control is constructed and the thread of execution comes back to the container side.

Finishing Creating the Control

Once the control comes into existence, the container still has to set up a few more things. Remember, OLE controls are in-place objects, so the container needs to set up that part of the communication, as well as set up the necessary connections (like the event sink and the property notification sink).

The container is emerging from the call to the control's class factory's CreateInstance(), which is occurring within COleControlSite::CreateOrLoad().

At this point, the control may indicate through its status bits that it wants to have the container's COleControlSite right away. This is important if the control needs to do something such as work with the container's ambient properties. The container gets the control's miscellaneous status bits from the control by calling the control's IOleObject::GetMiscStatus(). The container checks to see if the

OLEMISC_SETCLIENTSITEFIRST bit is set. If that bit is set, then that means the control would like to be able to talk to the client site right away. This allows the control to synchronize with the container's ambient properties.

At this point, the container needs to let the control serialize its properties. First the container tries to initialize the control through IPersistMemory. The container asks the control for its IPersistMemory interface (by calling QueryInterface() through the control's IOleObject pointer). The container calls the control's IPersistMemory::Load() function using a temporary CMemFile object that was created way back in the container COccManager::CreateDlgControl(). The control creates a CArcive object out of the container's temporary CMemFile object and uses it to call the control's Serialize() function. COleControl::Serialize() calls the control's DoPropExchange() function to exchange persistent properties between the control and the container.

If the control doesn't implement IPersistMemory, the control container performs a QueryInterface() in the control to get the the control's IPersistStreamInit() interface so it can use a stream to save and store control properties. The OLE control sees only a CArcive object—it may be based on either a CMemFile or an OLE stream.

As a final step inside CreateOrLoad() (after the control's properties are serialized), the container hands its IClientSite interface over to the control (a requirement of being an in-place embedded item).

If COleControlSite::CreateOrLoad() succeeds, then the control is alive and breathing and almost ready to do its stuff. However, the container still needs to connect to the control so the control can (1) fire events to the container and (2) notify the container of property changes. This brings us to a very interesting aspect of OLE controls: OLE connections.

OLE Connections

OLE implements events through connection technology. OLE connections involve a special type of interface called a “connection point.” Normal OLE interfaces implement and expose the functionality of an OLE object. These normal interfaces are called incoming interfaces. A connection point implements interfaces going the other way: outgoing interfaces.

An OLE connection consists of two parts: the source (the object *calling* the connection interface) and the sink (the object *implementing* the connection interface). In the case of OLE control events, the OLE control is the source and the OLE control container is the sink. By exposing a connection point, an OLE control allows an OLE control container to establish connections to the control. Using the connection point mechanism, an OLE control obtains a pointer to a set of functions implemented by the control container.

A COleControl-derived class implements two connection points by default: one for events and one for property change notifications. The event connection is used for event firing, while the property change notification connection is used for notifying the control's container when a property value has changed. Of course, an OLE control is free to implement additional connection points.

OLE Connection Interfaces

OLE defines two interfaces for supporting connections: IConnectionPoint and IConnectionPointContainer. IConnectionPoint defines a single connection between an OLE control and its container, while IConnectionPointContainer "holds" connection points.

Here's the interface definition for IConnectionPointContainer:

```
interface IConnectionPointContainer {
public:
    virtual HRESULT EnumConnectionPoints(LPENUMCONNECTIONPOINTS* ppEnum) = 0;
    virtual HRESULT FindConnectionPoint(REFIID iid,
                                         LPCONNECTIONPOINT* ppCP) = 0;
};
```

Again, OLE controls maintain two connection points by default: the events and property notifications. Each of these connection points is represented by a separate object implementing IConnectionPoint.

The other connection interface is IConnectionPoint, which defines a single connection from one OLE object to another. Here's the interface definition for IConnectionPoint:

```
interface IConnectionPoint {
public:
    virtual HRESULT GetConnectionInterface(IID* pIID) = 0;
    virtual HRESULT GetConnectionPointContainer(
        IConnectionPointContainer** ppCPC) = 0;
    virtual HRESULT Advise(LPUNKNOWN pUnkSink,
                          DWORD* pdwCookie) = 0;
    virtual HRESULT Unadvise(DWORD dwCookie) = 0;
    virtual HRESULT EnumConnections(LPENUMCONNECTIONS* ppEnum) = 0;
};
```

IConnectionPointContainer and IConnectionPoint have the functions necessary to implement a connectable object. Let's examine how connections are established.

Establishing a Connection

An OLE control container can `QueryInterface()` the control for `IConnectionPointContainer` to find out if the control supports connections at all. If the control returns a valid `IConnectionPointContainer` pointer, the control container can then query the control for actual connections using `IConnectionPointContainer::FindConnectionPoint()`, asking for the interface by GUID. For example, if the container wants to set up a connection with the control's events interface, the container would call `FindConnectionPoint()` using the GUID representing the control's events interface.

Remember, the container implements the events interface that the control expects to see. Once the container has found the correct connection, the container can plug its interface pointer into the control's event set by calling `IConnectionPoint::Advise()`. `IConnectionPoint::Advise()` stores the interface so the control can use it later. For example, if the control container connects to the control and plugs its implementation of `IDispatch` into `IConnectionPoint::Advise()`, the control can call the container's `IDispatch::Invoke()` function.

MFC and Connections

Because OLE controls written in MFC do work, MFC must be implementing `IConnectionPoint` and `IConnectionPointContainer` somewhere. MFC implements `IConnectionPointContainer` in the undocumented class `COleConnPtContainer`, found in `OLECONN.CPP`. MFC implements `IConnectionPoint` in the `CConnectionPoint` class, also inside the file `OLECONN.CPP`. Let's see how an MFC OLE control connects to an MFC OLE container by getting back to following the creation of an OLE control.

We left off inside `COleControlSite::CreateControl()`. `COleControlSite::CreateOrLoad()` has just finished breathing life into an OLE control, and now the control and the container need to connect themselves. To set up the event connection, the `COleControlSite` asks the control for its event interface GUID using `IPProvideClassInfo2::GetGUID()` and asking for the GUID representing the control's primary event set. The container uses a constant named `GUIDKIND_DEFAULT_SOURCE_DISP_IID`, which is defined in the standard header files. When the control sees this, it knows to hand back the GUID for the events interface.

It's now the container's job to somehow implement `IDispatch` so that the control can use it to inform the container of certain events. This particular `IDispatch` interface is often called an event sink. In MFC, the event sink is implemented in `COleControlSite`. `COleControlSite`'s event sink is a nested member class called `m_xEventSink`. This class is a regular `IDispatch` interface (with `GetTypeInfo()`, `GetTypeInfoCount()`, `GetIDsOfNames()`, and `Invoke()`). However, as we'll soon see, it's implemented a bit differently from a standard MFC dispatch interface. `COleControlSite::CreateControl()` uses the member function `ConnectSink()` to connect

the container's events IDispatch to the control. ConnectSink() passes the control site's dispatch interface for the event sink, the member class `m_xEventSink`.

ConnectSink() retrieves the control's IConnectionPointContainer interface (using `QueryInterface()`, of course). If the container can retrieve the control's IConnectionPointContainer interface, then the container knows that the control supports connections. Next, ConnectSink() uses the control's `IConnectionPointContainer::FindConnection()` to find the connection point for the event set. This brings us one more set of MFC maps: the connection maps.

Connection Maps

MFC's connection maps are remarkably similar to the interface maps we saw in Chapter 11. Remember that the interface maps are MFC's way of implementing multiple interfaces within a single C++ class using nested classes and a table-driven `QueryInterface()` mechanism. MFC's support for COM is divided neatly into two sets of macros.

The first set of macros (`BEGIN_INTERFACE_PART/END_INTERFACE_PART`) defines the nested classes within the single `CCmdTarget`-derived class. You add this pair of macros for each interface implemented by the COM class. After adding the nested class macros, you prototype the interface functions. Running this through the preprocessor yields a `CCmdTarget`-derived class that implements multiple interfaces through nested classes.

The second set of macros (`DECLARE_INTERFACE_MAP`, `BEGIN_INTERFACE_MAP/END_INTERFACE_MAP`, and `INTERFACE_PART`) defines the lookup table used by `QueryInterface()`. MFC's interface map correlates a GUID to the nested class's vtable (which is what the client expects to see as an interface). Every time you use `INTERFACE_PART`, you add an `AFX_INTERFACEMAP_ENTRY` structure to the message map. The `CCmdTarget`'s `QueryInterface()` looks in the interface map searching for the interface's GUID, returning the vtable if it finds the interface.

MFC's connection maps work almost the same way. You can use the macros to add OLE connections to any `CCmdTarget`-derived class.

MFC has two macros (`BEGIN_CONNECTION_PART` and `END_CONNECTION_PART`) that define a nested class representing the connection. Here are the macros in the raw. Notice how similar they are to `BEGIN_INTERFACE_PART` and `END_INTERFACE_PART`:

```
#define BEGIN_CONNECTION_PART(theClass, localClass) \
class X##localClass : public CConnectionPoint \
{ \
public: \
    X##localClass() \
    { m_nOffset = offsetof(theClass, m_x##localClass); }
```

```
#define END_CONNECTION_PART(localClass) \
} m_x##localClass; \
friend class X##localClass;
```

MFC's support for nested classes requires that you prototype the interface functions. Connection parts work the same way: you prototype the functions you want in your connection. If you want to override any CConnectionPoint member functions or add member functions of your own, declare them between these two macros. For example, MFC defines a connection point for control events inside COleControl:

```
// Connection point for events
BEGIN_CONNECTION_PART(COleControl, EventConnPt)
    virtual void OnAdvise(BOOL bAdvise);
    virtual REFIID GetIID();
END_CONNECTION_PART(EventConnPt)
```

Adding these macros to the COleControl class defines a nested class representing the event connection point with two functions: OnAdvise() and GetIID().

MFC also defines a similar connection point for property notifications inside COleControl:

```
// Connection point for property notifications
BEGIN_CONNECTION_PART(COleControl, PropConnPt)
    CONNECTION_IID(IID_IPropertyNotifySink)
END_CONNECTION_PART(PropConnPt)
```

The CONNECTION_ID macro expands to the following:

```
REFIID GetIID() { return iid; }
```

Setting up the connection points using the macros is only half the story. MFC has to implement a lookup mechanism. Yep—another set of mapping macros: DECLARE_CONNECTION_MAP, BEGIN_CONNECTION_PART/END_CONNECTION_MAP, and CONNECTION_PART.

Whereas the interface map macros build up a table of AFX_INTERFACEMAP_ENTRY structures, the connection map macros construct a list of AFX_CONNECTIONPOINT_ENTRY structures:

```
struct AFX_CONNECTIONMAP_ENTRY
{
    const void* piid;
    size_t nOffset;
};
```

The AFX_CONNECTIONMAP_ENTRY structure maps the connection interface ID to the address of the connection interface's vtable. MFC uses this list to look up an OLE control's connection points.

So, MFC implements connection points through connection maps. Let's get back to following how a control and a control container establish a connection.

Establishing a Connection (Continued)

When we left off establishing the connection between the OLE control and the container, the container was about to plug its event sink into the control's connection. The control was trying to find the connection point using the control's IConnectionPointContainer::FindConnectionPoint() to find the event set's connection point.

The control's FindConnectionPoint() searches for the appropriate connection. If the container is asking for the event's connection, then FindConnectionPoint() takes a shortcut and immediately returns the event's connection point. Otherwise, FindConnectionPoint() searches the connection map for a connection with a matching interface GUID.

If the container finds the correct connection, the container plugs its event sink into the control by calling the connection point's Advise() function. On the control side, the control's connection point adds the interface (for example, the events IDispatch or the property notification sink) to a list of interfaces it maintains. Because the connection point maintains multiple interfaces, OLE controls can in a sense broadcast events to multiple control sites.

Once the container connects its events IDispatch, the container connects its property notify sink to the control.

Next, COleControlSite::CreateControl() calls COleControlSite::DoVerb() to in-place-activate the control. DoVerb() simply calls the control's IOleObject::DoVerb() function.

At this point, the control is up and running, talking to the container, and ready to fire events to its container. Now let's see how those OLE control events work.

OLE Control Events

So what is an OLE control event? We need to define that before MFC's OLE control container support makes any sense. An OLE control event is the notification a control's host receives when interesting things happen within a control. For example, imagine you have an OLE control that collected data from a real-time data source (that the New York Stock Exchange, for example). You could use the control inside

your application to monitor the changes in the market. The control would use an event to notify your program about stock price changes.

Adding events to an OLE control using ClassWizard is almost trivial. Just push the Add Event button, decide on a name for your event, and add any necessary parameters. ClassWizard adds a function to your control. Whenever you want to notify the container of the event, just call the function inside your control.

For example, consider the stock market control again. One interesting event might be when the price of a certain stock changes. To add that event to the stock control, you'd first pull up ClassWizard and push the Add Event button. Typing "CurrentPriceChanged" in the Event Name field causes ClassWizard to add the function FireCurrentPriceChanged() to your control. In addition, you might want to tell the container both the old price and the new price. To do that, just add parameters to the event. ClassWizard adds a function named FireCurrentPriceChanged(): Whenever you want to tell the container about the event, just call FireCurrentPriceChanged(), which magically signals the control's host about the event.

The OLE control event mechanism is actually quite interesting, so let's take a look at it.

If you understand the way Windows callback functions work, then you're already well on your way to understanding OLE control events. Consider for a moment the way a Windows event works. Windows just sits there waiting for things to happen—things like mouse movements and key presses. Whenever something interesting happens, Windows wants to tell the applications it's hosting about the event. To do that, each application running under Windows registers a callback function—the window procedure (sometimes called a message handler)—for each of its windows. Whenever an event belongs to a particular window (for instance, a mouse click occurs within the bounds of a window), Windows generates a message and calls that window's message handler using the message. The key idea here is that each application supplies a function and registers it with Windows. Then Windows uses the function to notify the application of various events.

The basic idea behind OLE control events is the same as Windows callback functions. The OLE control event mechanism just happens to be a bit more flexible and powerful.

Here's a good way to think about events. OLE control containers implement functions to handle events. To invoke an event, an OLE control calls the event handler implemented by the container. Using the analogy of the regular Windows architecture, the message handler is really a big event sink that is continually called by an event source (in this case, Windows itself).

OLE control containers implement something similar to a callback function. However, we're in OLE Land now, so it's not a function—it's an interface. The control container implements an Automation interface and makes it available to the control. Then the control can notify the container of events through the interface.

Let's take a look at how MFC does this.

Firing Events

Every time you add an event, ClassWizard deposits a function in your code. In the stocks example, ClassWizard added the function FireCurrentPriceChanged():

```
void FireCurrentPriceChanged(double dOldPrice, double dNewPrice);
```

FireCurrentPriceChanged() is really just a friendly wrapper around COleControl::FireEvent(). FireEvent() takes a dispatch ID and a pointer to an array of bytes representing the event parameters. FireEvent() then uses va_start() to set up a variable argument list, then calls the variable argument list-enabled function for firing events: FireEventV(). COleControl::FireEventV() gets a dispatch interface representing the control's events from the control.

The control knows about the container's events dispatch interface because the connection was set up when the control was created. Let's watch how MFC handles OLE control events.

MFC OLE Control Events

OLE controls fire events by calling functions in an interface implemented by the client. FireEventV() has to retrieve the container's interface. Here's where the OLE connections come in.

MFC's COleControl class has a member class of type CConnectionPoint called m_xEventConnPt. To invoke the event, FireEventV() first needs to get the event handler function from the container. FireEventV() uses m_xEventConnPt's GetConnections() function to get the connections. CConnectionPoint maintains an array of interface pointers. The CConnectionPoint holds the connection interface array. When the container created the control, the container asked the control for the connection point representing the events interface. Then the container plugged in the events interface using IConnectionPoint::Advise().

In MFC, the standard event connections happen to be dispatch interfaces (or IDispatch pointers). To execute the container's event, FireEventV() declares a COleDispatchDriver object on the stack. COleDispatchDriver is a handy MFC class that wraps an IDispatch pointer, making it easier to access IDispatch methods and properties.

Next, for each of the connections represented by the control's m_xEventConnPt object, FireEventV() retrieves the container's IDispatch pointer from m_xEventConnPt. The connections are represented by an array of IDispatch pointers. FireEventV() attaches the IDispatch interface to the COleDispatchDriver object using COleDispatchDriver::Attach(). Once the dispatch driver is attached to the IDispatch pointer, FireEventV() can easily call the container's IDispatch using

COleDispatchDriver::InvokeHelper(), passing the event's dispatch ID and parameter list. After the control finishes invoking the control container's method, FireEventV() cleans up by detaching the COleDispatchDriver object from the container's IDispatch pointer.

So that's how OLE controls fire events. It's a symbiotic relationship between the control and the control container. The control can fire events only as long as the container has functions that represent the events. Obviously, the container is going to have some responsibilities. Let's see what those are.

How MFC Handles Events

Recall that a control's events dispatch interface is inside the COleControlSite class. COleControlSite calls its IDispatch for events an event sink. The event sink is simply a version of IDispatch implemented the MFC way. That is, COleControlSite uses the nested class/interface map approach. That means COleControlSite maintains an IDispatch vtable within the nested class called COleControlSite::XEventSink, and COleControlSite maintains an event sink member called m_xEventSink.

When an OLE control container creates a control, the container asks the control for its IConnectionPointContainer interface. If the OLE control container gets back a valid IConnectionPointContainer interface, the container knows the control supports connections. Next, the container asks the control for the GUID of its primary event set (using another interface implemented by the control: IPProvideClassInfo2). Once the control container has the events interface's GUID, the control can get the IConnectionPoint pointer that represents the events. Once the container has that, the container can register its event handler (the event sink—the container's events dispatch interface—COleControlSite::m_xEventSink) with the control using IConnectionPoint::Advise().

Remember, the event sink is really just another version of IDispatch. MFC implements the event sink through some macros that generate an Event Sink map (unbelievable, huh? . . . more maps and macros!).

Any CCmdTarget-derived class can have an event sink—just as every CCmdTarget can have a message map. A CCmdTarget's event sink map relates an event set's dispatch member to a member function inside the CCmdTarget-derived class.

Let's track the event through the container's event sink.

Whenever an OLE control invokes an event, the OLE control just calls IDispatch::Invoke() (somehow implemented by the container). If the container is written using MFC, the call ends up in COleControlSite::m_xEventSink::Invoke(). MFC's event sink, m_xEventSink::Invoke(), calls COleControlSite::OnEvent(), calling the control's parent window's OnCmdMsg(), which calls COccManager::OnEvent()

(an undocumented MFC class that manages OLE controls), finally ending up in CCmdTarget::OnEvent().

MFC implements a lookup mechanism to match OLE control events to C++ member functions for handling the events. A CCmdTarget's event sink map is like the other lookup mechanisms in MFC. As with the other event mechanisms, MFC includes the DECLARE_EVENTSINK_MAP and the BEGIN_EVENTSINK_MAP/END_EVENTSINK_MAP macros to set up the event sink map. The event sink map really just boils down to an array of event sink entry structures.

```
struct AFX_EVENTSINKMAP_ENTRY
{
    AFX_DISPMAP_ENTRY dispEntry;
    UINT nCtrlIDFirst;
    UINT nCtrlIDLast;
};
```

Notice that AFX_EVENTSINKMAP_ENTRY includes an AFX_DISPMAP_ENTRY. That's a clue that MFC's treating the event as another Automation method. The AFX_DISPMAP_ENTRY represents the method that is called as a result of firing the event.

CCmdTarget::OnEvent() uses CCmdTarget::GetEventSinkEntry() to look in the CCmdTarget-derived object's event sink map to find an event map entry. If OnEvent() finds an event sink entry, OnEvent() packages the parameters (remember, they're still represented by an array of VARIANTs) and uses CCmdTarget::CallMemberFunction() to invoke the event.

Though this is a fairly elaborate scheme, it works pretty well—and it meshes right in with MFC's command-routing mechanism.

Developer Tip: Adding an Event Sink to a View

Adding your own event sink in a non-dialog window isn't very hard—you just have to do everything that AppWizard and ClassWizard do. There are two things you have to do to be able to respond to OLE control events: (1) write a function to respond to the event and (2) set up an event sink. Neither task is difficult—you just have to do a little extra typing.

Adding a Function to Handle the Event

Remember how OLE control events work: Whenever a control is created, the control expects the control container to implement functions to respond to events. After all, an OLE control event just boils down to an OLE control calling a function inside its host. It's up to the control container to implement that function. In this case, the view needs to handle the event. That's easy enough, provided you know what the signature of the event handler is supposed to look like. How do you figure that out?

Learning the Function Signature

When you use the Dialog Editor to add an OLE control to your dialog box and wire up event handlers, the Dialog Editor is smart enough to look up the type information for the control's events. That way, ClassWizard inserts a function with the correct signature. But if you're not using ClassWizard, you need to figure out the function signature yourself.

You have a couple of options for figuring out the signature of the event handler. The quick way out is to just create a scratch dialog and place the control in the dialog box. Then use ClassWizard to add event handlers. ClassWizard adds member functions to your dialog with the correct signatures. You can look at the ClassWizard-generated functions to find out the signature of the function to add to your view. Of course, this adds a (perhaps useless) dialog box to your application, making both the code and the resources larger.

If you don't feel like adding a scratch dialog box to your application, you can use a browser that dissects a type library into its constituent elements. OLE2VW32.EXE, written by Charlie Kindel, is quite a good one. It comes with Visual C++.

OLE2VW32.EXE looks in the registry for all the OLE objects on your system. By now, many of those objects are OLE controls. Highlighting the OLE object causes OLE2VW32 to show the object's interfaces in the lower right corner of its window.

Double-clicking the IDispatch interface in the list box yields another dialog box showing the object's dispatch interfaces. OLE control events happen through dispatch interfaces.

Next, prototype the function in your header file. For example, if you want to handle the CurrentPriceChanged event in your view, add the following function to your view:

Then be sure to implement the event handler in your view:

```
void CTestctlsView::OnCurrentPriceChanged(double dOldPrice,
                                         double dNewPrice) {
    // Respond to the current price changing...
}
```

Once that's done, you need to create an event sink for your view and add the event handler to the event sink.

Setting Up the Event Sink Map

MFC implements event sinks using event sink maps. Like all other MFC lookup mechanisms, the event sink map is created using macros. In this case, the macros are `DECLARE_EVENTSINK_MAP`, `BEGIN_EVENTSINK_MAP`, and `END_EVENTSINK_MAP`.

Use the `DECLARE_EVENTSINK_MAP` macro in your view's header file. For example, if your view is called `CTestctlsView`, you might define the class something like this:

```
class CTestctlsView : public CView {
...
// Some private methods and attributes
...
public:
...
// Some public methods and attributes
...
protected:
    DECLARE_EVENTSINK_MAP()
};
```

Then use `BEGIN_EVENTSINK_MAP` and `END_EVENTSINK_MAP` in your view's source file.

For example, this is what you might put inside your view's file:

```
BEGIN_EVENTSINK_MAP(CTestctlsView, CView)
// The event sink entries will go here.
END_EVENTSINK_MAP()
```

Finally, you need to add event handlers to your event map. Naturally, there's a macro for doing that as well: it's called `ON_EVENT()`. This macro takes five parameters: (1) the class for which the event sink map is intended, (2) the control's

ID, (3) the dispatch ID of the event, (4) the actual event handler function, and (5) the signature of the event handler.

Where do all these parameters come from? The first two are easy. You already know the name of the view and you added the control's ID. If you generated a dummy dialog box, you can get the dispatch ID from the ON_EVENT macros entered by ClassWizard. You can get the dispatch ID from OLE2VW32 as well. The dispatch ID is displayed in the lower right window (the one with the FUNCDESC caption). Just check for the memid (which stands for member ID) field. The event function is the handler you defined. Finally, the signature is a series of VTS_ constants that represent Automation data types. You can find the VTS_ constants inside the MFC header file AFXDISP.H.

For example, to add the OnCurrentPriceChanged() function to your event sink, here's what you'd add to the event sink map:

```
ON_EVENT(CTestctlsView, ID_STOCKSCTL, 1,
         OnCurrentPriceChanged, VTS_R8 VTS_R8)
```

This event belongs to CTestctlsView, has an ID represented by ID_STOCKSCTL, has a dispatch ID of 1, uses OnCurrentPriceChanged() as the event-handling function, and has an argument list containing two double numbers.

Whenever the stock control fires the current-price-changed event, the view picks up the event via its event sink map, which ends up calling CTestctlsView::OnCurrentPriceChanged(). You can implement OnCurrentPriceChanged() to reflect the new data. Perhaps you'd like to update a ticker display or a graph.

Before leaving OLE controls, let's examine one other interesting OLE control feature: property pages.

Inside OLE Control Property Pages

Because OLE controls are Automation objects, they expose properties via IDispatch. That means their clients can access the control's properties very easily. In addition to exposing automation properties in the normal way, OLE controls provide a way for the client to change the properties by implementing some tabbed dialog boxes that let the user access the control's properties. Programming an OLE control's property page is fairly easy—especially when using Visual C++'s ClassWizard. However, if you just follow Microsoft's instructions, you'll miss some of the beauty of OLE controls and a chance to see OLE in action as a real integration technology.

Let's take a look inside OLE control property pages so we can see what's happening behind the scenes. In the process, we'll see how OLE is making its mark in the area for which it was intended: software components.

Property Pages in a Nutshell

OLE controls implement property pages so that clients can modify a control's properties without having to implement any user-interface code. For example, if there wasn't such a thing as property sheets, anyone who wanted to write a client to manipulate your control's properties would have to spend a lot of time rigging up dialog boxes and so forth. OLE controls are supposed to be self-contained packages of software. Why should an OLE control client have to implement a lot of user-interface code to set up a control's properties? The control knows all about its own properties, so it makes sense to let the control be responsible for providing some user interface for accessing properties. OLE controls provide this user interface through a verb.

Because OLE controls are OLE document content objects, they are able to implement verbs. OLE controls implement several verbs, one of them being the Properties verb. OLE controls usually react to the Properties verb by displaying a dialog box with tabs for manipulating a control's various properties. Figure 15-2 illustrates Microsoft's grid control (inserted in the test container provided with Visual C++). The figure shows how Microsoft's grid control responds to the properties verb by displaying this type of tabbed dialog box.

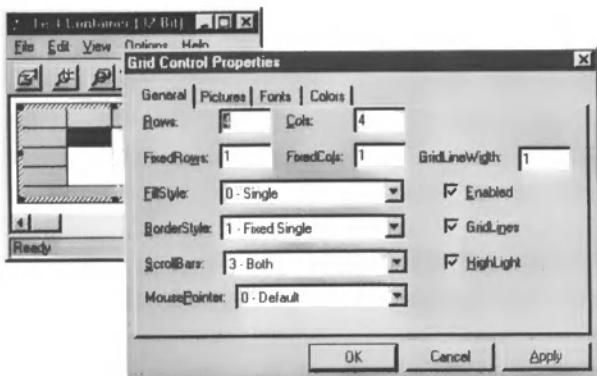


Figure 15-2. The properties of Microsoft's grid control

Before dissecting properties, let's take a closer look at Figure 15-2's screen shot. There are six OLE objects showing. First, the OLE control container (the test container) implements some OLE interfaces, so it's an OLE object. Second, there's the grid control itself. Finally, notice the four tabs in the dialog box. Each tab is a separate OLE object. This is a great illustration of OLE as an integration technology.

Notice how the test container, the grid control, and the properties dialog box appear to work as a single, cohesive unit. If you didn't know better, you might think this was all happening within a single EXE file (as in the old, pre-OLE days). But it's not! The test container is one EXE file. Of course, the grid control resides inside a

DLL. The dialog box tab that's showing (the "General" tab) also resides inside the control's DLL. Finally, the objects behind the other three tabs (the "Pictures," "Fonts," and "Colors" tabs) come from MFC40.DLL, which comes with Visual C++. Wow—smooth, seamless, and standard integration of three separate binary files! This is one of the main goals of OLE. Let's see how it all works.

Programming the Property Page

Start by examining what it takes to set up this machinery using Visual C++. Working with OLE control property pages in Visual C++ is surprisingly easy. It involves three steps: (1) defining the OLE control property, (2) setting up a dialog box template for the property page, and (3) linking the properties between the dialog box and the control itself. Now let's take a look at each of these steps.

Defining the OLE Control Property

The first step is to define a property within the OLE control. If you're using Visual C++, it's no problem. Just use ClassWizard to add the Automation property in the normal way. Make sure you're looking at the "OLE Automation" tab, then press the Properties button and add the property. It's important to remember the external name because it's critical for implementing the property page. For example, if you're writing an OLE control for monitoring a real-time data source, you may want to have the control beep whenever the data changes. Furthermore, you may want to provide a way for turning the sound on and off. The most straightforward way to do this is to add an Automation property called "SOUND" to the control.

Setting Up the Dialog Template

The second step is to set up the dialog box template for the property page. Each property page is really just another dialog box. In fact, the ControlWizard generates a blank default property page whenever you create the control. You can use this dialog box or create a new one if you want. Add a control to the dialog box for manipulating the OLE control's property. With the sound property, you'd probably use a checkbox in the dialog box. So far, so good.

Linking the Dialog Box Control to the OLE Control's Property

Now that there's a checkbox in the dialog box, you need to link the state of the checkbox (on or off) to the OLE control's property. Again, ClassWizard makes this almost trivial. Use ClassWizard to add a member variable to the dialog box representing the checkbox. Do this by clicking on the "Member Variables" tab and pressing the "Add Variable" button. ClassWizard then asks you some information about the variable using a dialog box like the one shown in Figure 15-3.

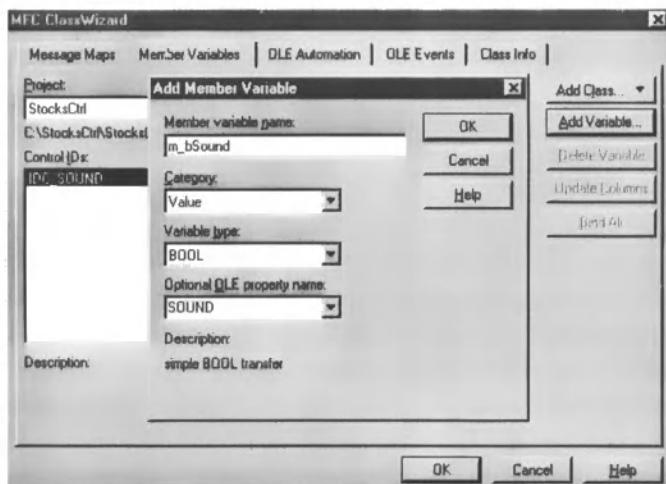


Figure 15-3. Adding a member variable to the property page dialog box

This adds a member variable representing the checkbox to your property page dialog. In addition, ClassWizard sets up MFC's dynamic data exchange mechanism (DDX/DDV), which exchanges the state of the checkbox (checked or unchecked) and the member variable (`m_bSound`).

Also, notice the “Optional OLE property name” field, which contains the string “SOUND”. Recall that the OLE control’s sound property (the regular Automation property) is called SOUND. If you guessed that the OLE control’s sound property and the state of the checkbox in the property page are linked via Automation, then you’re right. Remember that the external name of the control’s sound property is SOUND. In effect, the property page becomes an Automation client to the control, driving the control’s `IDispatch` interface. Here’s how that works.

Inside the Properties Verb

When an OLE control is embedded within some sort of container, the container usually has some means of invoking the control’s verbs. For example, if you’ve added a control to Microsoft’s test container, you can get to the Verbs menu by moving the mouse cursor over the control’s border and pressing the right mouse button. This gives you one of those tracking menus. If the control supports property pages, you’ll see the Properties verb in that menu. Selecting the Properties verb tells the in-place item to invoke its Properties verb. (The OLE control is an embedded item that exposes the `IOleObject` interface, which the container uses to invoke verbs by calling `IOleObject::DoVerb()`.)

OLE controls are OLE document content objects, so they implement `IOleObject`, which is used by the client to invoke verbs. That is, the client calls the control's `IOleObject::DoVerb()` function. Inside the control, the framework routes the container's `IOleObject::DoVerb()` call to `COleControl::OnDoVerb()`, which ends up in `COleControl::OnProperties()`.

`COleControl::OnProperties()` works in the following way. The control's properties are linked via `IDispatch`. Each property page is a separate OLE object and is going to communicate with the OLE control through the control's main `IDispatch` pointer. That means the control has to hand its `IDispatch` pointer over to the property pages so they can use it to manipulate the control's properties. `COleControl::OnProperties()` retrieves the OLE control's main `IDispatch` pointer (the one through which the client can access the sound property, in this case). Because `COleControl` is ultimately derived from `CCmdTarget`, the control can extract its `IDispatch` pointer using `CCmdTarget::GetIDispatch()`. The OLE control will use this dispatch pointer in a moment to create the property pages.

Because each of these property pages is a separate OLE object, they're going to be created via the normal OLE method, by calling `CoCreateInstance()`. `CoCreateInstance()` uses an OLE object's GUID to create the OLE object. Of course, this means that an OLE control needs to know GUIDs of all the property pages it wants to show.

In keeping with tradition, MFC uses macros to manage the control's list of GUIDs representing property pages. These macros are called `DECLARE_PROPPAGEIDS`, `BEGIN_PROPPAGEIDS`, `END_PROPPAGEIDS`, and `PROPPAGEID`. You may have seen them in some OLE control source code. For example, the property page list for a control looks something like this. `DECLARE_PROPPAGEIDS` goes inside the control's header file:

```
DECLARE_PROPPAGEIDS(CStocksCtrl)
```

`BEGIN_PROPPAGEIDS`, `PROPPAGEID`, and `END_PROPPAGEIDS` go inside the control's source code file:

```
BEGIN_PROPPAGEIDS(CStocksCtrl, 1)
    PROPPAGEID(CStocksCtrlPropPage::guid)
END_PROPPAGEIDS(CStocksCtrl)
```

The macros simply expand to generate a static array of GUIDs. `DECLARE_PROPPAGEIDS` declares an array of GUIDs (representing property pages) and a member function so the framework can retrieve the list of GUIDs when necessary. `BEGIN_PROPPAGEIDS`, `PROPPAGEID`, and `END_PROPPAGEIDS` expand to implement the array of GUIDs.

After retrieving the OLE control's IDispatch, COleControl::OnProperties() calls the OLE API function OleCreatePropertyFrame() to create the dialog box that holds the property pages. COleControl::OnProperties() uses the OLE control's IDispatch pointer and list of GUIDs to create the dialog frame. OleCreatePropertyFrame() creates a modal dialog box that houses the tabbed property pages. Once you call it, OLE takes over and creates the property pages. Basically, OleCreatePropertyFrame() just calls CoCreateInstance() for each GUID in the list, setting up each property page in turn.

To see the rest of the property page story, let's jump to the other side of the fence and see how MFC implements OLE property pages. The class is COlePropertyPage.

Inside COlePropertyPage

In MFC, COlePropertyPage is the class used for handling OLE property pages. It's derived from CDialog, so it does everything a CDialog object does. In addition to regular dialog box support, COlePropertyPage also implements an interface called IPagination2. When you pull up an OLE control's property pages, the tabbed dialogs are placed within a dialog frame. The dialog frame uses each property page's IPagination2 interface to communicate with the property page.

After the property page client (in this case, the OLE control) calls OleCreatePropertyPage(), OLE starts calling CoCreateInstance() for each property page being managed by the OLE control. Provided that CoCreateInstance() succeeds, control is handed over to the COlePropertyPage's constructor. After the COlePropertyPage is constructed, OleCreatePropertyFrame() calls QueryInterface() on the property page to get its IPagination2 interface. Once OLE has the IPagination2 interface, OLE uses the interface to set up the property page.

Remember that the OLE control implements IDispatch (so it can expose its properties). So any client code that wants to access the OLE control's properties has to have a copy of the OLE control's IDispatch property. In other words, the OLE property page needs the OLE control's IDispatch pointer. Here's where the property page gets the control's IDispatch pointer. IPagination2 has a function named SetObjects(). IPagination2::SetObjects() takes an array of IUnknown pointers. In the case of a regular MFC-based OLE control, OleCreatePropertyFrame() passes the control's main IDispatch pointer to the property page. The OLE property page saves the IDispatch pointer for later use (for example, when the property page wants to access the OLE control's property).

Notice that both the property frame and IPagination2::SetObjects() deal with IUnknown pointers. So there's quite a bit of flexibility here. Though IDispatch is the most logical way for a control to deal with properties, you're not limited to using IDispatch. Those IUnknown pointers could be your own custom interface. Though

MFC uses IDispatch, there's no reason you couldn't write your own property page class. However, note that both the control and the property page have to agree on what type of interface to use.

Accessing the Properties

Once the OLE property page has the OLE control's IDispatch pointer, the property page can easily access the control's properties. Because the property page has a copy of the OLE control's main IDispatch pointer, it's no problem. MFC extends the DDX mechanism to handle OLE control property pages.

Using the previous example, the default property page performs this magic in the DoDataExchange() function. Here's the code that Wizard inserts:

```
void CStocksCtrlPropPage::DoDataExchange(CDataExchange* pDX)
{
    //{{AFX_DATA_MAP(CStocksCtrlPropPage)
    DDP_Check(pDX, IDC_SOUND, m_bSound, _T("Sound") );
    DDX_Check(pDX, IDC_SOUND, m_bSound);
    //}}AFX_DATA_MAP
    DDP_PostProcessing(pDX);
}
```

The two lines of code between the comments perform the data exchange (1) between the dialog boxes and (2) between the control and the dialog box's member variable. MFC provides a number of property exchange functions. Table 15-1 lists MFC's property page exchange functions.

Table 15-1. MFC's property exchange functions

Function Name	Links This Kind of Dialog Control to a Property
DDP_CBIndex:	The selected string's index in a combo box
DDP_CBString:	The selected string in a combo box (without an exact match)
DDP_CBStringExact:	The selected string in a combo box (exact match)
DDP_Check:	A checkbox
DDP_LBIndex:	The selected string's index in a list box
DDP_LBString:	The selected string in a list box (without an exact match)
DDP_LBStringExact:	The selected string in a list box (with an exact match)
DDP_Radio:	A radio button
DDP_Text:	Text

COlePropertyPage implements several member functions for performing the actual work: GetPropText()/SetPropText(), GetPropCheck()/SetPropCheck(), GetPropRadio()/SetPropRadio(), and GetPropIndex()/SetPropIndex(). They all work more or less the same way, except they exchange different types of data. As an example, take a look at how a COlePropertyPage exchanges data between a checkbox and an Automation property.

Inside GetPropCheck()

To get the data from the property page to the OLE control, GetPropCheck() simply has to use the OLE control's main IDispatch pointer to set the OLE control's property. COlePropertyPage maintains an array of IDispatch pointers. This array was filled with pointers when the property page was created (when the OLE control called OleCreatePropertyFrame(), which used IPropertyPage2::SetObjects()). As the first order of business, GetPropCheck() declares a COleDispatchDriver object on the stack. COleDispatchDriver is MFC's wrapper for IDispatch, which has helper functions for using an IDispatch interface.

GetPropCheck() then goes through each IDispatch pointer in the array trying to access the Automation property by name. GetPropCheck() does this by calling IDispatch::GetIDsOfNames() to get the property's DISPID. Once GetPropCheck() has the DISPID, GetPropCheck() attaches the IDispatch pointer to the COleDispatchDriver object on the stack and calls COleDispatchDriver::GetProperty() to retrieve the control's Automation properties.

SetPropCheck() works almost identically, except that the function sets the property in the OLE control instead of just reading the property.

Property Page Wrapup

Property pages are a fast and convenient way for users to access various attributes of an OLE object. Property pages are standard fare for OLE controls and come free if you use the ControlWizard and MFC to create your controls.

OLE controls expose their properties via IDispatch. Whenever an OLE control client wants to use a control's property page, the client just needs to invoke the Properties verb. As a result, the OLE control uses the OLE API function OleCreatePropertyFrame() to create a dialog box that holds a set of tabbed dialog boxes representing the control's various properties. The OLE control passes its main IDispatch pointer when calling OleCreatePropertyFrame(), which passes the IDispatch pointer on to the property page. Each property page caches the OLE control's IDispatch pointer and uses the pointer to access the OLE control's properties.

Property pages are a great example of how MFC works to smooth out the edges between EXE and DLL modules.

Conclusion

So there you have it: OLE controls. OLE controls are probably the most exciting area of OLE these days. Using OLE controls is one of the fastest ways to customize your user interface. In addition, Microsoft is incorporating OLE controls into their Internet strategy in a big way.

OLE controls implement nearly every OLE technology currently available:

- Automation technology
- In-place activation technology
- Persistence technology
- Connection technology
- Property page technology

By implementing all these technologies, OLE controls can find a place almost anywhere. You can plant them into Visual Basic applications, and you can even roll your own OLE control container now that MFC supports it well.

Though most of these features would require a ton of work to get OLE controls up and running, a lot of it is boilerplate code that would be the same no matter who writes the control. MFC really shines when it comes to OLE controls! MFC implements just the right amount of boilerplate code to get you going and provides hooks where you add your own specific touch.

Stay on the lookout. You'll see OLE controls in many applications now that the technology is maturing.

A

A Field Guide to the MFC Source Code

If you've ever been camping or hiking, you probably brought along a field guide to help acclimate you to the area, the natural inhabitants, and so on.

Exploring the 150,000+ lines of MFC code can sometimes be a confusing and, yes, even treacherous undertaking. To help guide you through the MFC jungle, we offer this field guide. This resource both teaches you how to explore MFC and also acts as a reference if you should ever find yourself inextricably lost in the bowels of MFC.

The field guide starts by reviewing some common MFC coding styles. Knowledge of these styles helps you understand what an isolated block of code is doing without having to search around for all of the declarations and other things. These coding styles can also serve as a rule of thumb for MFC classes that you that write, if you so desire. Next, we give some hints that make exploring easier. We even point out some tools that can act as your machete as you hack your way through the thick MFC undergrowth.

After covering the MFC code style and exploring hints, we give in-depth details about these aspects of the MFC source code:

- The MFC directory structure.
- MFC header files.
- MFC inline files.
- MFC resources.
- MFC source files.

MFC Coding Techniques

In this section we review some of the coding techniques that Microsoft uses when writing MFC code. Knowledge of the Microsoft coding style can both help increase your MFC internals learning speed and also help you learn to write better MFC classes (if you choose to adopt Microsoft's style, which we recommend).

Class Declaration Subsections

It doesn't take too many header files to realize that the MFC team is following a pretty tight standard for class declarations. Class declarations are organized around "key" comment blocks. These comments are used by the MFC team to split the class declaration into the following subsections, which are usually in this order:

- // Constructors
- // Attributes
- // Operations
- // Overrideables
- // Implementation

Here's a quick rundown of what each class declaration subsection comment means, and some examples of what you will find in that subsection:

- // Constructors—Obviously, C++ constructors live in this subsection, but not so obviously, MFC usually puts any other "initialization" type member functions in this section, such as Create().
- // Attributes—Documented data members live here, along with member functions that manipulate (set or get) member data. These member functions are sometimes called accessor functions.
- // Operations—Documented member functions that will be directly called by the MFC user or other MFC classes are placed here. In some cases, there are enough operations that Microsoft subdivides this subsection even further (for example, // Operations—Grid functions, // Operations—Graphing functions, and so on).
- // Overrideables—Functions that are meant to be overridden by derived classes are placed here (a.k.a. virtual).
- // Implementation—Our favorite section. In MFC everything in this section is considered an implementation detail and is thus undocumented! Start here if you want to find out what makes a class tick.

Variable Naming—Think Hungarian!

If you learned Windows programming reading the good old Petzold book, you already are very familiar with Hungarian notation. In a nutshell, it's a variable-naming convention that prefixes variable names with the type of the variable.

After the prefix, the next letter is capitalized, and the first letter of any new words is capitalized too. Table A-1 gives a quick recap of the variable name prefixes suggested for the basic C++ types, with examples.

Table A-1. Help with Hungarian

Type	Prefix	Example	Comment
char	c	cDirSeparator	
BOOL	b	bIsSending	
int	n	nVariableCount	
UINT	n	nMyUnsignedInt	
WORD	w	wListID	
LONG	l	lAxisRatio	
DWORD	dw	dwPackedmessage	
* (pointer)	p	pWnd	
FAR *	lp	lpWnd	
LPSTR	lpsz	lpszFileName	z indicates NULL terminated.
handle	h	hWnd	
callback	lpfn	lpfnHookProc	Pointer to a function.

Note: Some developers are religiously against Hungarian, but if you want to be a fast draw with the MFC internals, you should be able to recognize and decode Hungarian notation in your sleep.

MFC adds some twists to the Hungarian naming conventions. To start with, Microsoft prefixes each class with *C* and member data with *m_*, for example, *CObject* and *m_hWnd*. Also, there are “standard” prefixes for some of the more common MFC classes, as shown in Table A-2.

You will see different examples of these names throughout the book. When you see a variable with “*m_*” in any MFC source code, mental alarms should go off that what you are seeing is a class data member and not a local variable or an argument to the function.

Table A-2. MFC extensions to Hungarian

Class	Prefix	Example
CRect	rect	rectScroll
CPoint	pt	ptMouseClick
CSize	sz	szRectangle
CString	str	strFind
CWnd	Wnd	WndControl
CWnd *	pWnd	pWndDialog

Symbol-Naming Conventions

MFC also follows a strict set of conventions for naming resource symbols. Knowing the prefixes used and the ranges they cover can help you “decode” the different symbols without having to look them up. Table A-3 shows these.

Tech Note 35 gives more details on resource ranges; be sure to read it if you really want to learn the gory details about the symbol ranges.

Table A-3. MFC symbol conventions

Type	Prefix	Example	Range
Shared by multiple resources	IDR_	IDR_MAINFRAME	1–0x6FFF
Dialog resource	IDD_	IDD_ABOUT	1–0x6FFF
Dialog resource help context ID (for context-sensitive help)	HIDD_	HIDD_HELP_ABOUT	0x20001–0x26FF
Bitmap resource	IDB_	IDB_SMILEY	1–0x6FFF
Cursor resource	IDC_	IDC_HAND	1–0x6FFF
Icon resource			
Menu or toolbar command	ID_	ID_CIRCLE_TOOL	0x8000–0xDFFF
Command help context	HID_	HID_CIRCLE_TOOL	0x1800–0x1DFFF
Message box prompt	IDP_	IDP_FATALError	8–0xFFFF
Message box help context	HIDP_	HIDP_FATALError	0x30008–0x3DFFF
String resource	IDS_	IDS_ERROR12	1–0x7FFF
Control in dialog template	IDC_	IDC_COMBO1	8–0xFFFF

The Proof Is in the Pudding!

By understanding Microsoft’s conventions for MFC class declaration, variable naming, and symbol naming, you will be able to look at a block of code and magically

figure out some key points without digging any further. To illustrate the point, here's a short code segment from MFC's source code:

```
if (bIdleStatus && m_bStatusSet) {
    m_bStatusSet = FALSE;
    GetOwner()->SendMessage(WM_SETMESSAGESTRING, AFX_IDS_IDLEMESSAGE);
}
if (bResetTimer)
    KillTimer(ID_TIMER); m_bDelayDone = FALSE;
if (m_pToolTip != NULL)
    m_pToolTip->DestroyWindow(); delete m_pToolTip; m_pToolTip = NULL;
```

Looking at this code, you can already deduce the following:

- bIdleStatus—Either a local variable or an argument, and it is a BOOL.
- m_bStatusSet—A class data member that is a BOOL.
- AFX_IDS_IDLEMESS—A string resource.
- bResetTimer—A local variable or an argument, and it is a BOOL.
- m_bDelayDone—A class data member that is a BOOL.
- m_pToolTip—A class data member that is a pointer (probably to a tool tip class).

It's amazing what knowledge of Hungarian notation and other MFC conventions can tell you about the code without ever having seen the class or variable declarations. When you scan a block of code, see how fast you can decode these clues. After a couple thousand lines of MFC source, it becomes second nature.

Common MFC #defines

Knowing a couple of common #defines that are checked in the MFC code can help give you some clues about what a body of code is doing.

Here's some information on the more common symbols that are checked:

- _MAC—When you look through the MFC source code, remember that Microsoft supports MFC on the Macintosh through the Visual C++ cross-platform edition for the Macintosh. There is only one set of MFC source code for both platforms, so any Macintosh-specific code is marked with a _MAC ifdef.
- _AFX_NO_OCC_SUPPORT—OLE controls are not supported on the Macintosh, so they are turned off with the _AFX_NO_OCC_SUPPORT flag. This flag is a great marker for OLE control code if you are exploring OLE controls in MFC.
- _DEBUG—This symbol is set for debug builds and off for release builds.

- `_AFXDLL`—Used to separate DLL from LIB code (see Chapter 10).
- `_UNICODE`—Defined if Unicode support is built into MFC.

Granularity and Swap Tuning?!

In some of the MFC source files, you may notice that the `IMPLEMENT_DYNAMIC` calls do not match the contents of the file. An example is `VIEWCORE.CPP`. `VIEWCORE.CPP` contains mostly `CView` implementation code, but at the bottom of the file you will find this:

```
// IMPLEMENT_DYNAMIC for CView is in wincore.cpp for .OBJ granularity
//   reasons
IMPLEMENT_DYNAMIC(CSplitterWnd, CWnd)    // for swap tuning
IMPLEMENT_DYNAMIC(CCtrView, CView)
```

After scratching our heads for a while, we asked our MFC internals insider, Dean McCrory, what the heck is going on. Here's Dean's answer:

The term *swap tuning* in these cases is really a misnomer. We do swap tuning with an internal Microsoft tool that organizes pages in the DLLs so the working set of MFC is smaller (this is done after the DLLs are linked).

The reason the `IMPLEMENT_` macros are located in what appears to be random files is for “library granularity.” That’s what is meant by `.OBJ` granularity in those comments. We want MFC, when statically linked, to be a “pay as you go” system. The more MFC features you actually use, the more MFC code will be brought in. The less you use, the less MFC code that the linker will drag in.

For the most part, the linker/compiler conspire to do this. We compile with `/Gy` and link with `/opt:noref`. The `/Gy` tells the compiler to put each function in its own comdat (think of it as a way of putting each function into what appears to the linker as a separate `.OBJ` file). The `/opt:noref` tells the compiler to eliminate unreferenced comdats. This almost achieves the “pay as you go” goal—only referenced functions are included in the final image. The problem comes when you have non-comdat sections which refer to functions. This includes data like message maps (which contain addresses of message handler functions), and data like `CRuntimeClass` objects (which contain the address of a `CreateObject` function, which calls the constructor, which in turn refers to the vtable, which in turn is data which refers to every virtual function, which in turn may reference other functions which are probably not used . . .). The latter problem is the only one that matters for `DECLARE_SERIAL` and `DECLARE_DYNCREATE`. You might notice the examples you give are `DECLARE_DYNAMIC`. So what’s the problem that we are solving??

There are cases where we refer to the CRuntimeClass of a particular object—even CRuntimeClass objects which are only DECLARE_DYNAMIC. If the definitions for such CRuntimeClass objects are put into the same file as other code for that class, then you may also pull in data in that same .OBJ. If that data (a message map, perhaps) refers to functions in that class, you end up dragging all such functions, even though they are not ever referenced. They are referenced only by data which itself is not referenced. But because data is not in a comdat, and non-comdat sections cannot be eliminated, the linker is stuck. Result: more code than you asked for.

Take your CView example. IMPLEMENT_DYNAMIC(CView) is in WINFRM.CPP because CFrameWnd needs to refer to RUNTIME_CLASS(CView). Therefore anything that uses CFrameWnd needs the CRuntimeClass object for CView. If we left the definition of CView's CRuntimeClass in VIEWCORE.CPP (where you might expect to find it), then apps that just used CFrameWnd would be much larger than necessary. This would make small utility apps and MFC 1.0 style apps much bigger (for example, samples\mfc\hello\hello.exe when statically linked would become a lot bigger).

The way we determine what goes where is we pick a few scenarios (for example, the helloapp sample and the default AppWizard-generated application) and examine the MAP file after a link to determine if unnecessary stuff is being pulled in. You can never get it 100%, but it is surprising with just a few changes how granularity can be improved.

Now you can impress your code buddies at the water cooler with this piece of MFC source code trivia!

Tools for Exploring MFC

Even if you know Hungarian like the back of your hand, there are times when you have to bite the bullet and search around the MFC source code for what you're after. Here are some pointers to tools that we've found and used that can help make MFC explorations easier.

The Visual C++ Browser

The Visual C++ browser is pretty good at helping you surf the MFC source code. To get the most out of the browser, load up the MFC.BSC file, which lives on your Visual C++ CD-ROM in the \msdev\mfc\src directory.

Visual C++ Find in Files

Yes, sometimes you have to resort to brute force and just “grep” like mad for a certain symbol. The Visual C++ Find in Files works great for this. Just point it at the MFC source code and go to town.

A Visual C++ Syntax-Coloring Tip

You can customize the Visual C++ syntax coloring to automatically highlight every MFC keyword! This is a great tool for quickly differentiating your classes from MFC’s classes. Check out the file USERTYPE.DAT on the book’s diskette for details.

Commercial Products

The ACI ObjectViewer is supposed to be a good general C++ browser. (It has a Macintosh-ish interface, though.)

There’s also a shareware tool called IXFWIN (Interactive Cross Referencer for Windows) that is a very nice “interactive grep” utility (nicer than Find in Files), which we highly recommend.

For more info, contact the tool’s author on the World Wide Web at <http://www.kudonet.com/~ixfwin>, or by e-mail at ixfwin@kudonet.com.

We have provided a copy of IXFWIN for you to try on the book’s floppy.

Visual Extensions is a set of tools that plug right into Visual C++ (including better searching/grepping). You can get more information via email from ray@newton.apple.com.

A Guide to the MFC Source Code

Now let’s look at the layout, structure, and contents of the MFC source code. This section is a great reference to keep handy when you are spelunking in the MFC source code and need a quick reference.

MFC Directory Structure

The MFC directory tree is shown in Figure A-1.

Note: Your installation may be slightly different, depending on the setup options you choose. If you want to explore all of these directories, you’ll find them all on the Visual C++ CD-ROM.

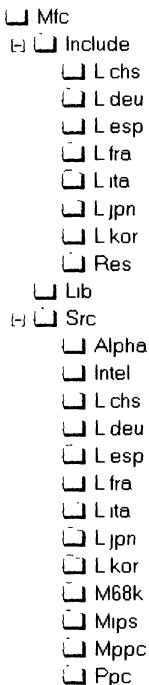


Figure A-1. The MFC directory tree

Here's a brief explanation of each directory:

- **Mfc**—The root directory. Nothing lives here; it's all in subdirectories.
- **Mfc\Include**—Contains the MFC header files (.H), resource files (.RC), inline files (.INL), Macintosh resources (.R), and MFC Help IDs (.HM).
- **Mfc\Include\L.chs**—Swedish resources.
- **Mfc\Include\L.deu**—German resources.
- **Mfc\Include\L.esp**—Spanish resources.
- **Mfc\Include\L.fra**—French resources.
- **Mfc\Include\L.ita**—Italian resources.
- **Mfc\Include\L.jpn**—Japanese resources.
- **Mfc\Include\L.kor**—Korean resources.
- **Mfc\Include\Res**—MFC cursors, bitmaps, and Macintosh cursors and bitmaps for inclusion in the resource (.RC) scripts.
- **Mfc\Lib**—Contains the prebuilt MFC libraries (.LIB) and also the program database files (.PDB) for debugging the libraries.

- Mfc\Src—Stores the MFC implementation, including MFC source files (.CPP), hidden headers (.H), resource scripts (.RC), and makefiles (.MAK).
- Mfc\Src\Alpha—The DLL .DEF/.PRF files for Alpha platforms (the mangling is different).
- Mfc\Src\Intel—The DLL .DEF/.PRF files for Intel platforms.
- Mfc\Src\L.chs—Swedish resources.
- Mfc\Src\L.deu—German resources.
- Mfc\Src\L.esp—Spanish resources.
- Mfc\Src\L.fra—French resources.
- Mfc\Src\L.ita—Italian resources.
- Mfc\Src\L.jpn—Japanese resources.
- Mfc\Src\L.kor—Korean resources.
- Mfc\Src\M68k—Macintosh Motorola 68K OLE stubs.
- Mfc\Src\Mips—The DLL .DEF/.PRF files for MIPS platforms.
- Mfc\Src\Mppc—The DLL .DEF/.PRF files for the Macintosh PowerPC and OLE stubs for that platform.
- Mfc\Src\Ppc—The DLL .DEF/.PRF files for the NT PowerPC build.

What's a .PRF File?

We asked Dean and his reply was:

A .PRF file is something we get out of our working set tuner. It contains a list of functions hit during our tuning scenario, organized to reduce the number of pages that are hit during the scenario. (An example scenario is booting a default appwiz mfc app, or booting wordpad, or using an OLE control in a dialog app, etc.) If you look at the functions in there, you'll see that the things at the beginning of the file are most likely hit often and the things at the bottom are hit less often. The .PRF file is fed to the linker so it can order the functions in the same order as in the .PRF. The .PRF name comes from "profiling" but in this case we are doing a different kind of profiling (working set profiling).

MFC Header Files

In this section we give two ways for you to look up the MFC header files. In the first section, we look at every header file (.H) and list what classes are in the header file. In the second section is a list of each MFC class and its corresponding MFC header file.

Header File Naming Convention

There is an interesting naming convention with MFC header files. Header files that end in an underscore (such as `afxplex_.h`) are intended to be included only by other mfc header files (the underscore is Microsoft's way of marking certain header files as "implementation", much like the `//Implementation` section in the headers themselves). Applications should avoid directly including headers that end in `_` as they may change massively or disappear in the future. Of course, this never stopped us!

A Map of MFC Header Files to Classes Declared

AFX.H—Non-GUI Classes

<code>CRuntimeClass</code>	<code>CObject</code>
<code>CException</code>	<code>CArchiveException</code>
<code>CFileException</code>	<code>CSimpleException</code>
<code>CMemoryException</code>	<code>CNotSupportedException</code>
<code>CFile</code>	<code>CStdioFile</code>
<code>CMemFile</code>	<code>CString</code>
<code>CTimeSpan</code>	<code>CTime</code>
<code>CFileStatus</code>	<code>CMemoryState</code>
<code>CArchive</code>	<code>CDumpContext</code>

AFXCMN.H—Windows Common Control Classes

<code>CToolInfo</code>	<code>CImageList</code>
<code>CDragListBox</code>	<code>CListCtrl</code>
<code>CTreeCtrl</code>	<code>CSpinButtonCtrl</code>
<code>CHolderCtrl</code>	<code>CSliderCtrl</code>
<code>CProgressCtrl</code>	<code>CHotKeyCtrl</code>
<code>CToolTipCtrl</code>	<code>CTabCtrl</code>
<code>CAimateCtrl</code>	<code>CToolBarCtrl</code>
<code>CStatusBarCtrl</code>	<code>CRichEditCtrl</code>

AFXCOLL.H—MFC Collections

<code>CByteArray</code>	<code>CWordArray</code>
<code>CDWordArray</code>	<code>CUIntArray</code>
<code>CPtrArray</code>	<code>COBArray</code>
<code>CPtrList</code>	<code>COBList</code>
<code>CMapWordToOb</code>	<code>CMapWordToPtr</code>
<code>CMapPtrToWord</code>	<code>CMapPtrToPtr</code>
<code>CStringArray</code>	<code>CStringList</code>
<code>CMapStringToPtr</code>	<code>CMapStringToOb</code>
<code>CMapStringToString</code>	

AFXCTL.H—OLE Control Classes

COleControlModule	CFontHolder
CPictureHolder	COleControl
COlePropertyPage	CPropExchange
CControlFrameWnd	

AFXVIEW.H—Control View Classes

CListView	CTreeView
-----------	-----------

AFXDAO.H—DAO Classes

CDaoException	CDaoRecordView
CDaoWorkspace	CDaoDatabase
CDaoRecordset	CDaoTableDef
CDaoQueryDef	CDaoFieldExchange
CDaoFieldCache	CDaoErrorInfo
CDaoWorkspaceInfo	CDaoDatabaseInfo
CDaoTableDefInfo	CDaoFieldInfo
CDaoIndexInfo	CDaoRelationInfo
CDaoQueryDefInfo	CDaoParameterInfo

AFXDB.H—ODBC Classes

CDBException	CFieldExchange
CDatabase	CRecordset
CRecordView	CRecordsetStatus
CFIELDINFO	

AFXDB_.H—CLongBinary Only

CLongBinary

AFXDD_.H—DDX/DDV Macro Declarations**AFXDISP.H—IDispatch and Class Factory Support**

CConnectionPoint	COleException
COleDispatchException	COleObjectFactory
COleTemplateServer	COleDispatchDriver
COleVariant	COleCurrency
COleDateTime	COleDateTimeSpan

AFXDLGS.H—Common Dialogs

CCommonDialog	CFindReplaceDialog
CFileDialog	CFontDialog
CColorDialog	CPrintDialog
CPageSetupDialog	CPropertySheet
CPropertyPage	

AFXDLL.H—MFC Extension DLL Declarations

CDynLinkLibrary

AFXDLLX.H—MFC DLL Helpers**AFXEXT.H—Enhanced UI Classes**

CBitmapButton	CControlBar
CStatusBar	CToolBar
CDialogBar	CSplitterWnd
CFormView	CEditView
CMetaFileDC	CRectTracker
CPrintInfo	CPrintPreviewState
CCreateContext	CControlBarInfo

AFXMSG.H—Age Map Signature Declarations**AFXMT.H—Multithreaded Class Declarations**

CSyncObject	CSemaphore
CMutex	CEvent
CCriticalSection	CSingleLock
CMultiLock	

AFXODLGS.H—OLE Common Dialogs

COleUILinkInfo	COleDialog
COleInsertDialog	COleConvertDialog
COleChangelconDialog	COlePasteSpecialDialog
COleLinksDialog	COleUpdateDialog
COleBusyDialog	COlePropertiesDialog
COleChangeSourceDialog	

AFXOLE.H—MFC OLE Support

COleDocument	COleLinkingDoc
COleServerDoc	CDocItem
COleClientItem	COleServerItem
COleDataSource	COleDropSource
COleDropTarget	COleMessageFilter
COleIPFrameWnd	COleResizeBar
COleStreamFile	COleDataObject

AFXPLEX.H—MFC Collection Memory Helpers

CPlex

AFXPRIV.H—MFC Private Classes: Undocumented

CSharedFile	CPreviewDC
CPreviewView	COleCntrFrameWnd
CMiniDockFrameWnd	CRecentFileList

CDockState	CDockContext
CArchiveStream	CDialogTemplate
CDockBar	COleControlLock

AFXRES.H—MFC Resource Symbol Definitions**AFXRICH.H—MFC Rich Text View Classes**

CRichEditView	CRichEditDoc
CRichEditCntrlItem	

AFXSOCK.H—WinSock Classes

CAsynchSocket	CSocket
CSocketFile	CSocketWnd

AFXSTAT_.H—Various MFC STATE Structure Definitions (AFX_STATE, AFX_THREAD_STATE, and so on)**AFXTEMPL.H—MFC Template Collection Classes****AFXTLS_.H—MFC Thread Local Storage Information**

CSimpleList	CTypedSimpleList
CThreadLocal	CProcessLocal

AFXV_CFG.H—Defines _AFX_PORTABLE for Portability across Compilers**AFXV_CPU.H—Declares/Defines CPU-Specific Stuff****AFXV_DLL.H—Special Header for MFC Extension DLLs****AFXV_MAC.H—Macintosh Definitions****AFXV_W32.H—Target/Version Control for Win32****AFXVER_.H—MFC Version Information****AFXWIN.H—MFC “Window” Classes**

CSize	CPoint
CRect	CResourceException
CUserException	CGdiObject
CPen	CBrush
CFont	CBitmap
CPalette	CRgn
CDC	CCClientDC
CWindowDC	CPaintDC
CMenu	CCmdTarget
CWnd	CDialog
CStatic	CButton
CListBox	CCheckListBox
CComboBox	CEdit
CScrollBar	CFrameWnd
CMDIFrameWnd	CMDIChildWnd

CMiniFrameWnd	CView
CScrollView	CWinThread
CWinApp	CDocTemplate
CSingleDocTemplate	CMultiDocTemplate
CDocument	CCmdUI
CDataExchange	CCCommandLineInfo
CDocManager	CCtrlView

WINRES.H—MFC Specific Windows Resource Declarations

MFC window messages, bitmap IDs, extended styles, virtual key codes, and so on.

A Map of MFC Classes to Header Files

CAnimateCtrl	AFXCMN.H	CArchive	AFX.H
CArchiveException	AFX.H	CArchiveStream	AFXPRIV.H
CAsynchSocket	AFXSOCK.H	CBitmap	AFXWIN.H
CBitmapButton	AFXEXT.H	CBrush	AFXWIN.H
CButton	AFXWIN.H	CByteArray	AFXCOLL.H
CCheckListBox	AFXWIN.H	CClientDC	AFXWIN.H
CCmdTarget	AFXWIN.H	CCmdUI	AFXWIN.H
CColorDialog	AFXDLGS.H	CComboBox	AFXWIN.H
CCCommandLineInfo	AFXWIN.H	CCommonDialog	AFXDLGS.H
CConnectionPoint	AFXDISP.H	CControlBar	AFXEXT.H
CCtrlBarInfo	AFXEXT.H	CCtrlFrameWnd	AFXCTL.H
CCreateContext	AFXEXT.H	CCriticalSection	AFXMT.H
CCtrlView	AFXWIN.H	CDBException	AFXDB.H
CDC	AFXWIN.H	CDWordArray	AFXCOLL.H
CDaoDatabase	AFXDAO.H	CDaoDatabaseInfo	AFXDAO.H
CDaoErrorInfo	AFXDAO.H	CDaoException	AFXDAO.H
CDaoFieldCache	AFXDAO.H	CDaoFieldExchange	AFXDAO.H
CDaoFieldInfo	AFXDAO.H	CDaoIndexInfo	AFXDAO.H
CDaoParameterInfo	AFXDAO.H	CDaoQueryDef	AFXDAO.H
CDaoQueryDefInfo	AFXDAO.H	CDaoRecordView	AFXDAO.H
CDaoRecordset	AFXDAO.H	CDaoRelationInfo	AFXDAO.H
CDaoTableDef	AFXDAO.H	CDaoTableDefInfo	AFXDAO.H
CDaoWorkspace	AFXDAO.H	CDaoWorkspaceInfo	AFXDAO.H
CDDataExchange	AFXWIN.H	CDatabase	AFXDB.H
CDialog	AFXWIN.H	CDialogBar	AFXEXT.H
CDialogTemplate	AFXPRIV.H	CDocItem	AFXOLE.H
CDocManager	AFXWIN.H	CDocTemplate	AFXWIN.H
CDockBar	AFXPRIV.H	CDockContext	AFXPRIV.H
CDockState	AFXPRIV.H	CDocument	AFXWIN.H
CDragListBox	AFXCMN.H	CDumpContext	AFX.H

CDynLinkLibrary	AFXDLL_.H	CEdit	AFXWIN.H
CEditView	AFXEXT.H	CEvent	AFXMT.H
CException	AFX.H	CFieldExchange	AFXDB.H
CFIELDINFO	AFXDB.H	CFile	AFX.H
CFileDialog	AFXDLGS.H	CFileException	AFX.H
CFileStatus	AFX.H	CFindReplaceDialog	AFXDLGS.H
CFont	AFXWIN.H	CFontDialog	AFXDLGS.H
CFontHolder	AFXCTL.H	CFormView	AFXEXT.H
CFrameWnd	AFXWIN.H	CGdiObject	AFXWIN.H
CHeaderCtrl	AFXCMN.H	CHotKeyCtrl	AFXCMN.H
CImageList	AFXCMN.H	CListBox	AFXWIN.H
CListCtrl	AFXCMN.H	CListView	AFXCVIEW.H
CLongBinary	AFXDB_.H	CMDIChildWnd	AFXWIN.H
CMDIFrameWnd	AFXWIN.H	CMapPtrToPtr	AFXCOLL.H
CMapPtrToWord	AFXCOLL.H	CMapStringToOb	AFXCOLL.H
CMapStringToPtr	AFXCOLL.H	CMapStringToString	AFXCOLL.H
CMapWordToOb	AFXCOLL.H	CMapWordToPtr	AFXCOLL.H
CMemFile	AFX.H	CMemoryException	AFX.H
CMemoryState	AFX.H	CMenu	AFXWIN.H
CMetaFileDC	AFXEXT.H	CMiniDockFrameWnd	AFXPRIV.H
CMiniFrameWnd	AFXWIN.H	CMultiDocTemplate	AFXWIN.H
CMultiLock	AFXMT.H	CMutex	AFXMT.H
CNotSupportedException	AFX.H	COBArray	AFXCOLL.H
COBList	AFXCOLL.H	COObject	AFX.H
COleBusyDialog	AFXODLGS.H	COleChangeIconDialog	AFXODLGS.H
COleChangeSourceDialog	AFXODLGS.H	COleClientItem	AFXOLE.H
COleCntrFrameWnd	AFXPRIV.H	COleControl	AFXCTL.H
COleControlLock	AFXPRIV.H	COleControlModule	AFXCTL.H
COleConvertDialog	AFXODLGS.H	COleCurrency	AFXDISP.H
COleDataObject	AFXOLE.H	COleDataSource	AFXOLE.H
COleDateTime	AFXDISP.H	COleDateTimeSpan	AFXDISP.H
COleDialog	AFXODLGS.H	COleDispatchDriver	AXDISP.H
COleDispatchException	AFXDISP.H	COleDocument	AFXOLE.H
COleDropSource	AFXOLE.H	COleDropTarget	AFXOLE.H
COleException	AFXDISP.H	COleIPFrameWnd	AFXOLE.H
COleInsertDialog	AFXODLGS.H	COleLinkingDoc	AFXOLE.H
COleLinksDialog	AFXODLGS.H	COleMessageFilter	AFXOLE.H
COleObjectFactory	AFXDISP.H	COlePasteSpecialDialog	AFXODLGS.H
COlePropertiesDialog	AFXODLGS.H	COlePropertyPage	AFXCTL.H
COleResizeBar	AFXOLE.H	COleServerDoc	AFXOLE.H
COleServerItem	AFXOLE.H	COleStreamFile	AFXOLE.H

COleTemplateServer	AFXDISP.H	COleUILinkInfo	AFXDLGS.H
COleUpdateDialog	AFXDLGS.H	COleVariant	AFXDISP.H
CPageSetupDialog	AFXDLGS.H	CPaintDC	AFXWIN.H
CPalette	AFXWIN.H	CPen	AFXWIN.H
CPictureHolder	AFXCTL.H	CPlex	AFXPLEX_.H
CPoint	AFXWIN.H	CPreviewDC	AFXPRIV.H
CPreviewView	AFXPRIV.H	CPrintDialog	AFXDLGS.H
CPrintInfo	AFXEXT.H	CPrintPreviewState	AFXEXT.H
CProcessLocal	AFXTLS_.H	CProgressCtrl	AFXCMN.H
CPropertyPage	AFXDLGS.H	CPropertySheet	AFXDLGS.H
CPropExchange	AFXCTL.H	CPtrArray	AFXCOLL.H
CPtrList	AFXCOLL.H	CRecentFileList	AFXPRIV.H
CRecordView	AFXDB.H	CRecordset	AFXDB.H
CRecordsetStatus	AFXDB.H	CRect	AFXWIN.H
CRectTracker	AFXEXT.H	CResourceException	AFXWIN.H
CRgn	AFXWIN.H	CRichEditCntrlItem	AFXRICH.H
CRichEditCtrl	AFXCMN.H	CRichEditDoc	AFXRICH.H
CRichEditView	AFXRICH.H	CRuntimeClass	AFX.H
CScrollBar	AFXWIN.H	CScrollView	AFXWIN.H
CSemaphore	AFXMT.H	CSharedFile	AFXPRIV.H
CSimpleException	AFX.H	CSimpleList	AFXTLS_.H
CSingleDocTemplate	AFXWIN.H	CSingleLock	AFXMT.H
CSize	AFXWIN.H	CSliderCtrl	AFXCMN.H
CSocket	AFXSOCK.H	CSocketFile	AFXSOCK.H
CSocketWnd	AFXSOCK.H	CSpinButtonCtrl	AFXCMN.H
CSplitterWnd	AFXEXT.H	CStatic	AFXWIN.H
CStatusBar	AFXEXT.H	CStatusBarCtrl	AFXCMN.H
CStudioFile	AFX.H	CString	AFX.H
CStringArray	AFXCOLL.H	CStringList	AFXCOLL.H
CSyncObject	AFXMT.H	CTabCtrl	AFXCMN.H
CThreadLocal	AFXTLS_.H	CTime	AFX.H
CTimeSpan	AFX.H	CToolBar	AFXEXT.H
CToolBarCtrl	AFXCMN.H	CToolInfo	AFXCMN.H
CToolTipCtrl	AFXCMN.H	CTreeCtrl	AFXCMN.H
CTreeView	AFXVIEW.H	CTypedSimpleList	AFXTLS_.H
CUIntArray	AFXCOLL.H	CUserException	AFXWIN.H
CView	AFXWIN.H	CWinApp	AFXWIN.H
CWinThread	AFXWIN.H	CWindowDC	AFXWIN.H
CWnd	AFXWIN.H	CWordArray	AFXCOLL.H

MFC Inline Files

Before we look at what lives where, let's answer an important question: What's an inline file? MFC toggles C++ inlining off for debug builds and on for release builds. MFC does this because inlined functions are very hard to debug because the compiler has done code substitution.

To accomplish inline toggling, MFC uses inline files, which have the INL extension. Each INL file has the INLINE functions for a corresponding header file (for example, AFX.H, AFX.INL). Instead of using the C++ keyword "inline", MFC uses _AFX_INLINE.

The fun begins when MFC is being used with _DEBUG turned on, which means that inlining will be turned off. In this scenario, _AFX_INLINE is defined to nothing. There are CPP files that #include the INL files if _DEBUG is defined. Then the CPP file is compiled as if it contained non-inlined source code.

In the case of a release build, where _DEBUG is not defined, the CPP files are not included in the build. Instead, _AFX_INLINE is defined to be the "inline" keyword, and the INL file is #included into the matching header, where it becomes good-inlined by the compiler.

Because inline files can be a little confusing at first, let's look at a quick example.: AFX.INL. The matching header file is AFX.H, and the matching CPP file is AFXINL1.CPP.

The following three lines from AFXVER_.H are the key:

```
#ifndef _DEBUG
    #define _AFX_ENABLE_INLINES
#endif
```

Now, in AFX.H we have this:

```
#ifdef _AFX_ENABLE_INLINES
    #define _AFX_INLINE inline
    #include <afx.inl>
#endif
```

In AFXINL1.CPP we have this:

```
#ifndef _AFX_ENABLE_INLINES
    #define _AFX_INLINE
    #include "afx.inl"
#endif
```

When MFC builds in _DEBUG mode, AFX_ENABLE_INLINES is not defined, so AFX.H does not include AFX.INL. However, AFXINL1.CPP does include AFX.INL, after #defining _AFX_INLINE to nothing.

When MFC builds in release mode (!_DEBUG_), AFX_ENABLE_INLINES is defined, so AFX.H includes AFX.INL after defining _AFX_INLINE to be “inline.” AFXINL1.CPP does not do anything in this case.

The MFC inline files usually match one to one with a corresponding header file, except for AFXWIN1.INL and AFXWIN2.INL. Both of these inline files match the AFXWIN.H header file. Microsoft split them up because they are pretty lengthy.

You will see the CPP-to-INL mapping in the MFC source file section.

MFC Resources

The MFC source code has a plethora of RC files lying around both in the “include” directory and the “src” directory. One great thing about the RC files is that you can open them up and take a peek with Visual C++. Some knowledge of what is in there does help, so here’s a look at what is inside each of the RC files:

- include\AFXCTL.RC—OLE control resources.
- include\AFXDB.RC—Database resources.
- include\AFXOLECL.RC—OLE client resources.
- include\AFXOLESV.RC—OLE server resources.
- include\AFXPRINT.RC—Printing resources.
- include\AFXRES.RC—Default resources.
- src\INDICATE.RC—AFX status bar indicator strings.
- src\MFCDB.RC—Defines the resources for the MFC DB DLL.
- src\MFCDLL.RC—Defines the resources for the MFC DLL, including most of the include*.RC files.
- src\MFCINTL.RC—Resources for the language-specific DLLs.
- src\MFCNET.RC—Resources for the MFC NET DLL.
- src\MFCOLE.RC—Resources for the MFC OLE DLL.
- src\PROMPTS.RC—Contains the string table for all of the standard MFC menu prompts.

Depending on the language ID, these RC files will include different strings from the language-specific subdirectories (for example, L.fra, L.deu, and so on).

If you have some spare time and want to have some fun, open up the MFCDLL.RC—it has most of the cool resources. See if you can figure out where all of them are used and why. Speaking of which . . .

A Mystery Resource!

Figure A-2 shows a very odd resource that lives in MFCDLL.RC.

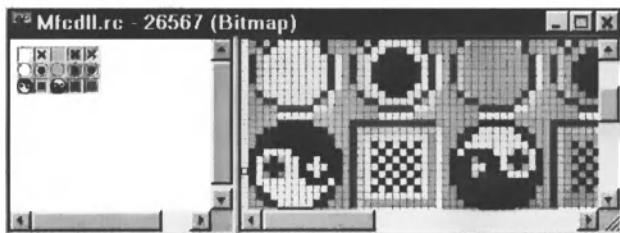


Figure A-2. The mystery graffiti resource

The resource is a bitmap with ID 26567. It contains what look like normal checkbox images for a three-state checkbox. But take a look at the bottom row. There are two funky yin-yang images!

Can you figure out why they are here? Perhaps the MFC team is working on a top-secret yin-yang MFC control class?

Actually, these states will never be hit by a three-state checkbox, so it looks as if whoever drew this bitmap had some fun with the empty spaces. It's MFC graffiti. Just think, every copy of Windows 95 ships with this bitmap in the MFC DLL (so does your application).

MFC Source Files

In this section we reveal what is included in every MFC source file! Just as in the include file section, we first look at the information from a file-to-class view and then from a class-to-file view, for your convenience.

Some of the MFC classes are conveniently implemented in one source file. The “bigger” MFC classes (CWnd, CFrameWnd, CWinApp) are spread across several files. This guide shows which file has the majority of a class’s implementation. You may still have to resort to a “grep utility” to find specific member functions. Also, don’t forget that some classes are entirely implemented in inline files, so include the INL files in any searches you perform.

A Map of MFC Source Files to Their Contents

AFXABORT.CPP	AfxAbort() helper
AFXASERT.CPP	ASSERT helper: AfxAssertFailedLine()
AFXCRIT.CPP	MFC internal thread helpers: for marking critical sections inside MFC
AFXDBCS.CPP	Static _afxDBCS symbol lives and is initialized here

AFXINL1.CPP	Inline file for AFX.INL, AFXCOLL.INL, AFXDLGS.INL, AFXEXT.INL
AFXINL2.CPP	Inline file for AFXWIN1.INL
AFXINL3.CPP	Inline file for AFXWIN2.INL
AFXMEM.CPP	MFC memory helpers: operator new, debug memory allocators and checkers, CMemoryState
AFXSTATE.CPP	MFC state helpers, constructors, and destructors (_AFX_THREAD_STATE and AFX_MODULE_STATE)
AFXTLS.CPP	CSimpleList, CThreadLocal
AFXTRACE.CPP	MFC TRACE helpers, including a map of all messages to strings
APP3D.CPP	_AFX_CTL3D_STATE helpers, constructor, destructor
APP3DS.CPP	CWinApp::Enable3dControlsStatic()
APPCORE.CPP	CWinApp "core," CCommandLineInfo
APPDLG.CPP	CWinApp "dialog"
APPGRAY.CPP	CWinApp gray background color
APPHELP.CPP	Context-sensitive help helpers
APPHELPX.CPP	CWinApp help
APPINIT.CPP	CWinApp initialization and helpers
APPMODUL.CPP	_AFX_TERM_APP_STATE constructor, and other things
APPPRNT.CPP	CWinApp printing
APPTERM.CPP	AfxWinTerm() helper; frees everything
APPUI.CPP	CWinApp user interface
APPUI2.CPP	More CWinApp UI
APPUI3.CPP	Still more CWinApp UI
ARCCORE.CPP	CArchive "core," CRuntimeClass
ARCEX.CPP	CArchiveException
ARCOBJ.CPP	CArchive/CObject interface
ARRAY_B.CPP	CByteArray
ARRAY_D.CPP	CDWordArray
ARRAY_O.CPP	CObArray
ARRAY_P.CPP	CPtrArray
ARRAY_S.CPP	CStringArray
ARRAY_U.CPP	CUIIntArray
ARRAY_W.CPP	CWordArray
AUXDATA.CPP	AUX_DATA() constructor and destructor
BARCORE.CPP	CControlBar "core"
BARDLG.CPP	CDialogBar
BARDOCK.CPP	CDockBar, CMiniDockFrameWnd
BARSTAT.CPP	CStatusBar
BARTOOL.CPP	CToolBar
CCDATA.CPP	Common control class helper routines
CMDTARG.CPP	CCmdTarget, CCmdUI

CTLCACHE.CPP	COleControl cache
CTLCONN.CPP	COleControl connection
CTLCORE.CPP	COleControl "core"
CTLDATA.CPP	COleControl "data"
CTLEVENT.CPP	COleControl "event"
CTLFONT.CPP	CFontHolder
CTLFRAME.CPP	CCtrlFrameWnd
CTLINL.CPP	OLE control inline file (AFXCTL.INL)
CTLINPLC.CPP	COleControl "in-place activation"
CTLINTL.CPP	_WEP for a resource-only DLL
CTLILIC.CPP	AfxVerifyLicFile() helper
CTLMODUL.CPP	COleControlModule
CTLOBJ.CPP	COleControl::XOleObject()
CTLPBAG.CPP	CBlobProperty, COleControl::XPeristPropertyBag
CTLPICT.CPP	CPictureHolder
CTLPPG.CPP	COlePropertyPage
CTLPROP.CPP	COleControl "property"
CTLPROPX.CPP	CPropExchange
CTLPSET.CPP	COleControl "property set," CPropsetPropExchange
CTLPSTG.CPP	COleControl::XPersistStorage
CTLPSTM.CPP	COleControl::XPersistStreamInit
CTLREFL.CPP	CReflectorWnd, CParkingWnd
CTLREG.CPP	OLE registration helpers
CTLTRACK.CPP	COleControl "tracker"
CTLVIEW.CPP	COleControl::XViewObject
DAOCORE.CPP	CDaoDatabase, CDaoErrorInfo, CDaoException, CDaoFieldInfo, CDaoIndexInfo, CDaoParameterInfo, CDaoQueryDef, CDaoQueryDefInfo, CDaoRecordset, CDaoRelationInfo, CDaoTableDef, CDaoTableDefInfo, CDaoWorkspace, and CDaoWorkspacelInfo
DAODFX.CPP	CDaoFieldCache, CDaoFieldExchange
DAOVIEW.CPP	CDaoRecordView
DBCORE.CPP	CDBException, CDatabase, CRecordset
DBFLT.CPP	Record exchange helpers (RFX_*)
DBLONG.CPP	CLongBinary
DBRFX.CPP	CFieldExchange
DBVIEW.CPP	CRecordView
DCMETA.CPP	CMetaFileDC
DCPREV.CPP	CPreviewDC
DLGCLR.CPP	CColorDialog
DLGCOMM.CPP	CCCommonDialog and helpers
DLGCORE.CPP	CDialog "core"

DLGDATA.CPP	CDataExchange and DDX/DDV helpers
DLGFILE.CPP	CFileDialog
DLGFLOAT.CPP	Floating point DDX/DDV helpers
DLGFNT.CPP	CFontDialog
DLGFR.CPP	CFindReplaceDialog
DLGPRNT.CPP	CPageSetupDialog, CPrintDialog
DLGPROP.CPP	CPropertyPage, CPropertySheet
DLGTEMPL.CPP	CDialogTemplate
DLLDB.CPP	Special DLL code for ODBC/DAO
DLLINIT.CPP	CDynLinkLibrary, extension DLL helpers
DLLMODUL.CPP	DLL module
DLLNET.CPP	Initializes MFC extension DLL
DLLOLE.CPP	Special OLE code for DLLs
DOCCORE.CPP	CDocument, CMirrorFile
DOCKCONT.CPP	CDockContext
DOCKSTAT.CPP	CCtrlBarInfo, CDockState, CDockBar, and CControlBar
DOCMAPI.CPP	CDocument MAPI support
DOCMGR.CPP	CDocManager
DOCMULTI.CPP	CMultiDocTemplate
DOCSINGL.CPP	CSingleDocTemplate
DOCTEMPL.CPP	CDocTemplate
DUMPCONT.CPP	CDumpContext
DUMPFLT.CPP	CDumpContext for floats
DUMPINIT.CPP	MFC dumping initialization
DUMPOUT.CPP	MFC dump output helpers.
EXCEPT.CPP	CException, CSimpleException, CMemoryException, CNotSupportedException, and CUserException
FILECORE.CPP	CFile
FILELIST.CPP	CRecentFileList
FILEMEM.CPP	CMemFile
FILESHRD.CPP	CSharedFile
FILEST.CPP	CFileStatus
FILETXT.CPP	CStdioFile
FILEX.CPP	CFileException
LIST_O.CPP	COBList
LIST_P.CPP	CPtrList
LIST_S.CPP	CStringList
MAP_PP.CPP	CMapPtrToPtr
MAP_PW.CPP	CMapPtrToWord
MAP_SO.CPP	CMapStringToOb
MAP_SP.CPP	CMapStringToPtr

MAP_SS.CPP	CMapStringToString
MAP_WO.CPP	CMapWordToOb
MAP_WP.CPP	CMapWordToPtr
MTCORE.CPP	CSyncObject
MTEX.CPP	CSemaphore, CMutex, CEvent, CSingleLock, and CMultiLock
NOLIB.CPP	#pragmas that run off libraries
OBJCORE.CPP	CObject
OCCCONT.CPP	CWnd OLE control containment
OCCDDX.CPP	DDX routines for OLE controls
OCCDDXF.CPP	DDX float routines for OLE controls
OCCDLG.CPP	COccManager
OCCEVENT.CPP	CCmdTarget OLE control code
OCCLOCK.CPP	COleControlLock
OCCMGR.CPP	COccManager "core"
OCCSITE.CPP	COleControlSite
OLEBAR.CPP	COleResizeBar
OLECALL.CPP	_AfxDispatchCall assembly routine for IDispatch (yucky!)
OLECLI1.CPP	COleClientItem
OLECLI2.CPP	COleFrameHook, COleClientItem
OLECLI3.CPP	COleClientItem
OLECONN.CPP	CConnectionPoint, COleConnPtContainer, and CEnumConnPoints
OLEDATA.CPP	OLE helpers
OLEDISP1.CPP	COleDispatchException
OLEDISP2.CPP	COleDispatchDriver
OLEDLGS1.CPP	COleChangelconDialog, COleConvertDialog, COleDialog, COleInsertDialog, COleLinksDialog, COlePasteSpecialDialog, COleUpdateDialog, and COleUllLinkInfo
OLEDLGS2.CPP	COleBusyDialog, COleDialog
OLEDLGS3.CPP	COleChangeSourceDialog, COleDialog, and COlePropertiesDialog
OLEDLL.CPP	OLE DLL helpers
OLEDOBJ1.CPP	COleDataObject
OLEDOBJ2.CPP	COleDataSource
OLEDOC1.CPP	CDocItem, COleDocument
OLEDOC2.CPP	COleDocument
OLEDROP1.CPP	COleDropSource
OLEDROP2.CPP	COleDropTarget
OLEENUM.CPP	CEnumArray
OLEEXP.CPP	OLE DLL helpers
OLEFACT.CPP	COleObjectFactory
OLEINIT.CPP	OLE_DATA and OLE initialization helpers
OLEIPFRM.CPP	COleCntrFrameWnd, COleIPFrameWnd

OLELINK.CPP	COleLinkingDoc
OLELOCK.CPP	OLE app exit helpers
OLEMISC.CPP	COleException
OLEMSGF.CPP	COleMessageFilter
OLEPRO32.CPP	OLE property helpers
OLEPSET.CPP	CProperty, CPropertySet
OLEREG.CPP	OLE registration helpers
OLESTRM.CPP	COleStreamFile
OLESVR1.CPP	COleServerDoc
OLESVR2.CPP	COleServerItem
OLETSVR.CPP	COleTemplateServer
OLETYPLB.CPP	CCmdTarget IDispatch routines, CTypeLibCache
OLEUI1.CPP	COleClientItem, COleDocument
OLEUI2.CPP	COleDocument
OLEUNK.CPP	MFC IUnknown helpers
OLEVAR.CPP	COleCurrency, COleDateTime, COleTimeSpan, and COleVariant
OLEVERB.CPP	CEnumOleVerb
PLEX.CPP	CPlex
PPGCOLOR.CPP	CColorButton, CColorPropPage
PPGFONT.CPP	CFontPropPage
PPGPICT.CPP	CPicturePropPage
PPGSTOCK.CPP	CStockPropPage
SOCKCORE.CPP	CAsyncSocket, CSocket, CSocketWnd, CSocketFile
STDAFX.CPP	Used for precompiled headers
STRCORE.CPP	CString
STREX.CPP	CString advanced operators
THRDCORE.CPP	CWinThread
TIMECORE.CPP	CTime, CTimeSpan
TOOLTIP.CPP	CToolTipCtrl, CWnd tooltip code, and CToolInfo
TRCKRECT.CPP	CRectTracker
VALIDADD.CPP	AFX memory helpers: AfxIsValidAddress()
VIEWCMN.CPP	CLListView, CTreeView
VIEWCORE.CPP	CView, CCtrlView
VIEWEDIT.CPP	CEditView
VIEWFORM.CPP	CFormView
VIEWPREV.CPP	CPreviewView, CPrintPreviewState, CView print preview
VIEWPRNT.CPP	CPrintingDialog, CPrintInfo, CView printing
VIEWRICH.CPP	CRichEditView, CRichEditDoc, CRichEditCntrlItem
VIEWSCRL.CPP	CScrollView
WINBTN.CPP	CBitmapButton
WINCORE.CPP	CWnd "core"

WINCTRL1.CPP	CStatic, CButton, CListBox, CComboBox, CEdit, and CScrollBar
WINCTRL2.CPP	CDragListBox, CSplitButtonCtrl, CSliderCtrl, CProgressCtrl, CHeaderCtrl, CHotKeyCtrl, CAnimateCtrl, CTabCtrl, CTreeCtrl, CListCtrl, CToolBarCtrl, CStatusBarCtrl, CImageList, and CArchiveStream
WINCTRL3.CPP	CCheckListBox
WINCTRL4.CPP	CRichEditCtrl
WINFRM.CPP	CFrameWnd core
WINFRM2.CPP	CFrameWnd docking
WINFRMX.CPP	CWnd/CFrameWnd help support
WINGDI.CPP	CDC, CClientDC, CWindowDC, CPaintDC, CPen, CBrush, CFont, CBitmap, CPalette, CRgn, and CGdiObject
WINGDIX.CPP	CDC utility members, some GDI helpers
WINHAND.CPP	Internal MFC CHandleMap
WINMAIN.CPP	AfxWinMain()
WINMDI.CPP	CMDIFrameWnd, CMDIChildWnd
WINMENU.CPP	CMenu
WINMINI.CPP	CMiniFrameWnd
WINOCC.CPP	CWnd OLE control members
WINSPLIT.CPP	CSplitterWnd
WINSTR.CPP	String helpers
WINUTIL.CPP	Window helpers

A Map of MFC Classes to Source Files

CAnimateCtrl	WINCTRL2.CPP
CArchive	ARCCORE.CPP, ARCOBJ.CPP
CArchiveException	ARCEX.CPP
CArchiveStream	WINCTRL2.CPP
CAsyncSocket	SOCKCORE.CPP
CBlobProperty	CTLBAG.CPP
CBitmap	WINGDI.CPP
CBitmapButton	WINBTN.CPP
CBrush	WINGDI.CPP
CButton	WINCTRL1.CPP
CByteArray	ARRAY_B.CPP
CCheckListBox	WINCTRL3.CPP
CClientDC	WINGDI.CPP
CCmdTarget	CMDTARG.CPP, OCCEVENT.CPP, OLETYPLB.CPP
CCmdUI	CMDTARG.CPP
CCColorButton	PPGCOLOR.CPP
CCColorPropPage	PPGCOLOR.CPP
CCColorDialog	DLGCLR.CPP

CComboBox	WINCTRL1.CPP
CCommandLineInfo	APPCORE.CPP
CCommonDialog	DLGCOMM.CPP
CConnectionPoint	OLECONN.CPP
CControlBar	BARCORE.CPP, DOCKSTAT.CPP
CControlBarInfo	DOCKSTATE.CPP
ControlFrameWnd	CTLFRMAE.CPP
CCreateContext	100% inline
CCriticalSection	100% inline
CCtrlView	VIEWCORE.CPP
CDBException	DBCORE.CPP
CDC	WINGDI.CPP, WINGDIX.CPP
CDWordArray	ARRAY_D.CPP
CDaoDatabase	DAOCORE.CPP
CDaoDatabaseInfo	DAOCORE.CPP
CDaoErrorInfo	DAOCORE.CPP
CDaoException	DAOCORE.CPP
CDaoFieldCache	DAODFX.CPP
CDaoFieldExchange	DAODFX.CPP
CDaoFieldInfo	DAOCORE.CPP
CDaoIndexInfo	DAOCORE.CPP
CDaoParameterInfo	DAOCORE.CPP
CDaoQueryDef	DAOCORE.CPP
CDaoQueryDefInfo	DAOCORE.CPP
CDaoRecordView	DAOVIEW.CPP
CDaoRecordset	DAOCORE.CPP
CDaoRelationInfo	DAOCORE.CPP
CDaoTableDef	DAOCORE.CPP
CDaoTableDefInfo	DAOCORE.CPP
CDaoWorkspace	DAOCORE.CPP
CDaoWorkspacelInfo	DAOCORE.CPP
CDataExchange	DLGDATA.CPP
CDatabase	DBCORE.CPP
CDialog	DLGCORE.CPP
CDialogBar	BARDLG.CPP
CDialogTemplate	DLGTMPL.CPP
CDocItem	OLEDOC1.CPP
CDocManager	DOCMGR.CPP
CDocTemplate	DOCTEMPL.CPP
CDockBar	BARDOCK.CPP, DOCKSTAT.CPP
CDockContext	DOCKCONT.CPP

CDockState	DOCKSTAT.CPP
CDocument	DOCCORE.CPP, DOCMAPI.CPP
CDragListBox	WINCTRL2.CPP
CDumpContext	DUMPCONT.CPP, DUMPFILT.CPP
CDynLinkLibrary	DLLINIT.CPP
CEdit	WINCTRL1.CPP
CEditView	VIEWEDIT.CPP
CEnumArray	OLEENUM.CPP
CEnumConnPoints	OLECONN.CPP
CEnumOleVerb	OLEVERB.CPP
CEvent	MTEX.CPP
CException	EXCEPT.CPP
CFieldExchange	DBRFX.CPP
CFieldInfo	100% inline
CFile	FILECORE.CPP
CFileDialog	DLGFILE.CPP
CFileException	FILEX.CPP
CFileStatus	FILEST.CPP
CFindReplaceDialog	DLGFR.CPP
CFont	WINGDI.CPP
CFontDialog	DLGFNT.CPP
CFontHolder	CTLFONT.CPP
CFontPropPage	PPGFONT.CPP
CFormView	VIEWFORM.CPP
CFrameWnd	WINFRM.CPP, WINFRM2.CPP, WINFRMX.CPP
CGdiObject	WINGDI.CPP
CHeaderCtrl	WINCTRL2.CPP
CHotKeyCtrl	WINCTRL2.CPP
CLImageList	WINCTRL2.CPP
CLListBox	WINCTRL1.CPP
CLListCtrl	WINCTRL2.CPP
CLListView	VIEWCMN.CPP
CLongBinary	DBLONG.CPP
CMDIChildWnd	WINMDI.CPP
CMDIFrameWnd	WINMDI.CPP
CMapPtrToPtr	MAP_PP.CPP
CMapPtrToWord	MAP_PW.CPP
CMapStringToOb	MAP_SO.CPP
CMapStringToPtr	MAP_SP.CPP
CMapStringToString	MAP_SS.CPP
CMapWordToOb	MAP_WO.CPP

CMapWordToPtr	MAP_WP.CPP
CMemFile	FILEMEM.CPP
CMemoryException	EXCEPT.CPP
CMemoryState	AFXMEM.CPP
CMenu	WINMENU.CPP
CMetaFileDC	DCMETA.CPP
CMiniDockFrameWnd	BARDOCK.CPP
CMiniFrameWnd	WINMINI.CPP
CMirrorFile	DOCCORE.CPP
CMultiDocTemplate	DOCMULTI.CPP
CMultiLock	MTEX.CPP
CMutex	MTEX.CPP
CNotSupportedException	EXCEPT.CPP
CObArray	ARRAY_O.CPP
CObList	LIST_O.CPP
CObject	OBJCORE.CPP
COccManager	OCCDLG.CPP, OCCMGR.CPP
COleBusyDialog	OLEDLGS2.CPP
COleChangeIconDialog	OLEDLGS1.CPP
COleChangeSourceDialog	OLEDLGS3.CPP
COleClientItem	OLECLI1.CPP, OLECLI2.CPP, OLECLI3.CPP, OLEUI1.CPP
COleCntrFrameWnd	OLEIPFRM.CPP
COleConnPtContainer	OLECONN.CPP
COleControl	CTLCACHE.CPP, CTLCONN.CPP, CTLCORE.CPP, CTLDATA.CPP, CTLEVENT.CPP, CTLFONT.CPP, CTLFRAME.CPP, CTLINPLC.CPPP, CTLINTL.CPP, CTLIC.CPP, CTLOBJ.CPP, CTLPBAG.CPP, CTLPCT.CPP, CTLPPG.CPP, CTLPROP.CPP, CTLPROPX.CPP, CTLPSET.CPP, CTLSTG.CPP, CTLSTM.CPP, CTLREFL.CPP, CTLREG.CPP, CTLTRACK.CPP, CTLVIEW.CPP
COleControlLock	OCCLOCK.CPP
COleControlModule	CTLMODUL.CPP
COleControlSite	OCCSITE.CPP
COleConvertDialog	OLEDLGS1.CPP
COleCurrency	OLEVAR.CPP
COleDataObject	OLEOBJ1.CPP
COleDataSource	OLEOBJ2.CPP
COleDateTime	OLEVAR.CPP
COleDateTimeSpan	OLEVAR.CPP
COleDialog	OLEDLGS1.CPP, OLEDLGS2.CPP, OLEDLGS3.CPP
COleDispatchDriver	OLEDISP2.CPP
COleDispatchException	OLEDISP1.CPP
COleDocument	OLEDOC1.CPP, OLEUI2.CPP, OLEDOC2.CPP, OLEUI1.CPP

COleDropSource	OLEDROP1.CPP
COleDropTarget	OLEDROP1.CPP
COleException	OLEMISC.CPP
COleFrameHook	OLECLI2.CPP
COleIPFrameWnd	OLEIPFRM.CPP
COleInsertDialog	OLEDLGS1.CPP
COleLinkingDoc	OLELINK.CPP
COleLinksDialog	OLEDLGS1.CPP
COleMessageFilter	OLEMSGF.CPP
COleObjectFactory	OLEFACT.CPP
COlePasteSpecialDialog	OLEDLGS1.CPP
COlePropertiesDialog	OLEDLGS3.CPP
COlePropertyPage	CTLPPG.CPP
COleResizeBar	OLEBAR.CPP
COleServerDoc	OLESVR1.CPP
COleServerItem	OLESVR2.CPP
COleStreamFile	OLESTRM.CPP
COleTemplateServer	OLETSVR.CPP
COleUILinkInfo	OLEDLGS1.CPP
COleUpdateDialog	OLEDLGS1.CPP
COleVariant	OLEVAR.CPP
CPageSetupDialog	DLGPRNT.CPP
CParkingWnd	CTLREFL.CPP
CPaintDC	WINGDI.CPP
CPalette	WINGDI.CPP
CPen	WINGDI.CPP
CPictureHolder	CTLPICT.CPP
CPicturePropPage	PPGPICT.CPP
CPlex	PLEX.CPP
CPoint	100% inline
CPreviewDC	DCPREV.CPP
CPreviewView	VIEWPREV.CPP
CPrintDialog	DLGPRNT.CPP
CPrintInfo	VIEWPRNT.CPP
CPrintPreviewState	VIEWPREV.CPP
CProcessLocal	100% inline
CProgressCtrl	WINCTRL2.CPP
CProperty	OLEPSET.CPP
CPropertyPage	DLGPROP.CPP
CPropertySheet	DLGPROP.CPP
CPropertySet	OLEPSET.CPP

CPropExchange	CTLPROPX.CPP
CPropsetPropExchange	CTLPSET.CPP
CPtrArray	ARRAY_P.CPP
CPtrList	LIST_P.CPP
CRecentFileDialog	FILELIST.CPP
CRecordView	DBVIEW.CPP
CRecordset	DBCORE.CPP
CRecordsetStatus	100% inline
CRect	100% inline
CRectTracker	TRCKRECT.CPP
CReflectorWnd	CTLREFL.CPP
CResourceException	100% inline
CRgn	WINGDI.CPP
CRichEditCntrlItem	VIEWRICH.CPP
CRichEditCtrl	WINCTRL4.CPP
CRichEditDoc	VIEWRICH.CPP
CRichEditView	VIEWRICH.CPP
CRuntimeClass	ARCCORE.CPP
CScrollBar	WINCTRL1.CPP
CScrollView	VIEWSCRL.CPP
CSemaphore	MTEX.CPP
CSharedFile	FILESHRD.CPP
CSimpleException	EXCEPT.CPP
CSimpleList	AFXTLS.CPP
CSingleDocTemplate	DOCSINGL.CPP
CSingleLock	MTEX.CPP
CSize	100% inline
CSliderCtrl	WINCTRL2.CPP
CSocket	SOCKCORE.CPP
CSocketFile	SOCKCORE.CPP
CSocketWnd	SOCKCORE.CPP
CSpinButtonCtrl	WINCTRL2.CPP
CSplitterWnd	WINSPLIT.CPP
CStatic	WINCTRL1.CPP
CStatusBar	BARSTAT.CPP
CStatusBarCtrl	WINCTRL2.CPP
CStdioFile	FILETXT.CPP
CStockPropPage	PPGSTOCK.CPP
CString	STRCORE.CPP, STREX.CPP
CStringArray	ARRAY_S.CPP
CStringList	LIST_S.CPP

CSyncObject	MTCORE.CPP
CTabCtrl	WINCTRL2.CPP
CThreadLocal	AFXTLS.CPP
CTime	TIMECORE.CPP
CTimeSpan	TIMECORE.CPP
CToolBar	BARTOOL.CPP
CToolBarCtrl	WINCTRL2.CPP
CToolInfo	TOOLTIP.CPP
CToolTipCtrl	TOOLTIP.CPP
CTreeCtrl	WINCTRL2.CPP
CTreeView	VIEWCMN.CPP
CTypeLibCache	OLETYPLB.CPP
CUIntArray	ARRAY_U.CPP
CUserException	EXCEPT.CPP
CView	VIEWCORE.CPP, VIEWPRNT.CPP
CWinApp	APPCORE.CPP, APPDLG.CPP, APPGRAY.CPP, APPHELPX.CPP, APPINIT.CPP, APPRNT.CPP, APPUI.CPP, APPUI2.CPP, APPUI3.CPP
CWinThread	THRDCORE.CPP
CWindowDC	WINGDI.CPP
CWnd	WINCORE.CPP, TOOLTIP.CPP, WINFRMX.CPP, WINOCC.SPP, OCCCONT.CPP
CWordArray	ARRAY_W.CPP

MFC Hidden Implementation Headers

In the `Mfc\Src` directory, there are some header files that “hide” implementation-specific details. Those files are these:

- `AFXIMPL.H`—`AUX_DATA`, Macintosh byte-swapping helpers
- `BUILD_.H`—MFC build number (used by Microsoft)
- `COMMIMPL.H`—Common control implementation helpers
- `CTLIMPL.H`—OLE control undocumented implementation helpers and classes:
`CControlFrameWnd`, `CReflectorWnd`, `CParkingWnd`, `CPropertySection`, `CPropertySet`,
`CArchivePropExchange`, `CPropsetPropExchange`, `CStockPropPage`, `CCColorButton`,
`CCColorPropPage`, `CPicturePropPage`
- `DAOIMPL.H`—`_AFX.DAO.STATE`
- `DBIMPL.H`—`AFX_ODBC_CALL`, `_AFX.DB.STATE`
- `ELEMENTS.H`—`CString` helpers for collections and `CPlex`
- `OCCIMPL.H`—`COleControlContainer`, `COleControlSite`, `COccManager`
- `OLEIMPL.H`—OLE helpers and state objects

- OLEIMPL2.H—COleFrameHook, CTypeLibCache, CEnumArray, COleDispatchImpl, OLE_DATA, _AFX_OLE_STATE
- SOCKIMPL.H—_AFX_SOCK_STATE
- STDAFX.H—Master MFC include file that includes all other MFC includes based on preprocessor settings
- WINHAND_.H—CHandleMap declaration

Happy Trails . . .

Now that we've shown you around the MFC source code, you should be ready to start exploring on your own. Of course, you will find coverage of many of these classes in the body of this book, but if you are out exploring on your own, you may want to keep your MFC source code field guide handy just in case the MFC jungle gets so thick that you need a helping hand.

Happy MFC exploring!

B

The **MFC Internals** Floppy

The *MFC Internals* floppy contains a README.TXT file that describes the disk's contents in detail. Highlights include:

- The Chapter 6 Objfun sample in the CHAP6 directory.
- The Chapter 11 CoMath sample in the CHAP11 directory. This sample is written three ways—using multiple inheritance, using nested classes, and using MFC.
- The Chapter 12 DNDEDIT sample, a drag-and-drop-enabled text editor in the CHAP12 directory.
- The Chapter 14 MATHSRVR sample using MFC to implement automation in the CHAP14 directory.
- Your very own copy of the MFC FAQ (in both Word and HTML formats).
- Demos and white papers of products we thought you might be interested in.

Also remember that you can find updates to this book's text and other news at this URL:

http://www.stingsoft.com/mfc_internals

Index

- Absolute time, 15
Abstract base class serialization, 173
Abstraction, 1–2
ACI ObjectViewer, 662
Add() function, 455, 458
AddPage() function, 247
AddRef() function, 457
Advanced document/view architecture, 295–331
Advanced memory diagnostics, 187–192
AFX (application framework), 7–11
_AfxAbortProc() procedure, 304
AfxAllocMemoryDebug() function, 190–92
AfxAssertFailedLine() global function, 184
AfxAssertValidObject() global function, 185–186, 196
AfxBeginThread() function, 417–419
AfxCallWndProc() function, 78, 85, 87
_AfxCbtFilterHook() function, 76–77
_AfxCheckDialogTemplate() helper function, 211
AfxCheckMemory() global helper function, 194
_AFX_CLASSINIT structures, 410
_AFX_COLOR_STATE static object, 181, 236
_AfxCommDlgProc() function, 230, 232–234
_AfxCopyStgMedium() function, 521
AfxControlBar class, 55–57
AfxCreateDC() utility function, 236
AfxDeferRegisterClass() function, 55–56, 249
AfxDlgProc() procedure, 209
AfxDlIIGetClassObject() function, 497–498, 632
AFXDLLs, 407–408
AfxDoForAllClasses() function, 196
AfxDoForAllObjects() function, 195
AfxDumpMemoryLeaks() function, 195
AfxFindMessageEntry() function, 87
AfxFindResourceHandle() function, 412–413
AfxFrameOrView class, 55–57
AfxGetApp() function, 51–52
AfxGetInstanceHandle() function, 51
AfxGetMainWnd() function, 92
AfxGetModuleState() function, 52
_AfxHandleActivate() function, 90
_AfxHandleInitDialog() function, 78
_AfxHandleSetCursor() function, 90
AfxHookWindowCreate() function, 76
AfxInitExtensionModule() function, 409–410
AfxMDIFrame class, 55–57
afxMemDF global variable, 178
_AfxMsgFilterHook() function, 57, 76
_AfxOleHookProc() procedure, 239
AfxOleLockApp() function, 493
AfxOleOnReleaseAllObjects() function, 494
_AfxOlePropertiesEnabled() function, 239
AfxOleRegisterHelper() function, 491–493
AfxOleUnlockApp() function, 493–494
AfxOutputDebugString() function, 183–184
_AfxPauseDisplayName() function, 239
_AfxStandardSubclass() function, 77
afxRegisteredClasses global variable, 55
AfxSetText() utility function, 220
_AfxThreadEntry() function, 425
_AFX_THREAD_STARTUP structure, 420–423
_AFX_THREAD_STATE class, 403–406
AfxThrowFileException() function, 147
AfxTrace() function, 180
AfxWinInit() function, 52–53, 56–57
AfxWinMain() function, 47–48, 51–52
AfxWnd class, 56–57
AfxWndProc() function, 38, 76–78, 85, 87–88
AFX_CHECKLIST_STATE class, 254, 255
AFX_CHECK_DATA pointer, 254
AFX_CLASSINIT structure, 157–158
AFX_CORE_STATE structure, 51
AFX_DATA macro, 153
AFX_DATACACHE_ENTRY structure, 519

- A**
- AFX_DATADEF macro, 153
 - AFX_EXCEPTION_LINK structure, 145
 - AFX_EXTENSION_MODULE structure, 408, 410–411
 - AFX_IDI_STD_FRAME icon, 56–57
 - AFX_IDI_STD_MDIFRAME icon, 56
 - AFX_ID_PANE_FIRST ID, 347
 - AFX_ID_PANE_LAST ID, 347
 - AFX_INTERFACEMAP_ENTRY structure, 477, 479, 637
 - AFX_MODULE_PROCESS_STATE definition, 400–401
 - AFX_MODULE_STATE definition, 401–403
 - AFX_MODULE_STATE structure, 48–52, 62, 399
 - AFX_MODULE_THREAD_STATE structure, 45
 - AFX_MSGMAP structure, 71–72, 74
 - AFX_MSGMAP_ENTRY structure, 70–75
 - AFX_OLE_STATE structure, 51
 - AFX_SIZEPARENTPARMS private structure, 352
 - AFX_WIN_STATE structure, 51, 493
 - AFX_ZERO_INIT_OBJECT macro, 207
 - Aggregation, 468–473
 - Alt-Backspace-Delete key combination, 19
 - AND_CATCH macro, 144
 - Animated controls, 19
 - API (Application Programming Interface), 7, 9
 - functions, 43
 - Application CWinApp-derived object, 86
 - Application data. *See* documents
 - Application frameworks, 7–10
 - classes, 12, 21–22
 - Application view, 86, 88
 - Application's document, 86
 - Applications
 - adding class factories, 494–495
 - basic Windows support, 29–30
 - boilerplate code, 30–32
 - command line parameters, 41
 - components, 30
 - copy of application name, 41
 - designating what causes it to fail, 52
 - document oriented, 25
 - entry point, 31
 - first window to receive messages, 18
 - getting single application object associated with, 51–52
 - handling data, 21
 - initialization, 32, 37, 42, 52–54
 - main application, 30
 - message handler, 32
 - message loop, 32
 - message pump, 39, 42
 - MFC (Microsoft Foundation Classes) vs C/SDK, 30–38
 - no messages in queue, 42
 - not document oriented, 25
 - notifying when event occurs, 17
 - objects, 39–42
 - on-line help, 22
 - print preview, 299
 - printing, 299
 - registering window classes, 54–57
 - retaining command line parameters, 41
 - run-time class information, 51
 - setting all documents, 54
 - showing main window, 54
 - shutdown and code clean-up, 42
 - starting parameters, 53
 - steps to qualify as Windows program, 31–32
 - toolbar, 23–24
 - windows, 30, 32, 39
 - WinMain() function, 36–37, 46–48
 - wrapping API functions, 43 - AppWizard, 262, 267, 291, 303
 - control bars, 366–367
 - OLE (Object Linking and Embedding), 434
 - Archiving exceptions, 13
 - Array retrieval members, 114
 - Arrays, 108–112
 - adding elements, 113
 - adding items to, 108
 - collections implemented as, 14
 - data members, 110
 - declaration for nontemplate classes, 109
 - growing, 110–112
 - initializing, 108
 - memory handling, 110–112
 - removing items, 109
 - retrieving data from, 109, 113–114
 - serialization, 173
 - ASSERT macro, 175–176, 184–185, 187
 - Assertions, 175–176, 184–187
 - ASSERT_VALID macro, 176, 185–186
 - Association, 122
 - Attach() function, 60
 - Automation, 25
 - history, 581–582
 - how it works, 584
 - late binding, 587
 - MFC and, 602–617
 - ODL (Object Description Language), 597–599
 - type information, 597–601
 - what it can do, 582–583
 - writing application, 584
 - Base classes, 17–18
 - DLL (Dynamic Link Library) base class for holding OLE controls, 26
 - exceptions, 13
 - OLE, 26
 - Basic application components, 39–54
 - CWinApp class, 39–42
 - CWnd class, 42–44
 - initializations to entire application, 53–54
 - state information, 48–51
 - turning window handles into window objects, 44–46
 - WinMain() function, 46–48, 51
 - BeginPaint() function, 20, 263
 - BEGIN_CONNECTION_PART macro, 637–638
 - BEGIN_EVENTSINK_MAP macro, 642, 644
 - BEGIN_INTERFACE_MAP() macro, 478–479, 637

- BEGIN_INTERFACE_PART() macro, 475–476, 495, 637
- BEGIN_MESSAGE_MAP macro, 71–74
- Binary files, 15
- Binary reuse, 468
- Bitmapped pushbutton, 19
- Bitmaps, 62
- Boilerplate code, 30–32
- Booch, Grady, 2
- Breakpoints, 184–185
- Brushes, 62
- Buffered I/O (input/output) files, 15
- Button list boxes, 19
- Bytes, 14
- C language, 4–5
- C++
 allocating objects, 446–447
 classes, 2, 5, 77, 436
 creation, 7
 debugger window, 175
 encapsulation, 2, 9
 exception example listing, 143
 global objects, 52
 header files, 6, 436
 inheritance, 2–3
 member functions, 3, 5
 message handling, 69–70
 objects, 4–6
 polymorphism, 3
 reuse of code and designs, 7
 RTTI (run-time type information)
 language, 154
 source code files, 6, 436
 strong typing, 9
 structs, 2, 5
 subset, 8–9
- C/SDK
 initializing particular instance of application, 37
 main window of application, 37
 message handling, 38, 65, 67–69
 message loop, 37
 showing window, 43
 typical window procedure listing, 68–69
 vs MFC (Microsoft Foundation Classes), 30–38
- WinMain() function, 36
- CallMemberFunction() function, 613
- CArchiveCtrl class, 19, 256
- Canned components, 10–11
- CArchive class, 13, 140, 163–165
 already-read objects arrays, 173
 data members, 166
 extraction and insertion operators for WORD, 167
- CArchive .ReadObject() member function, 168–169
- CArchive .WriteObject() function, 172
- CArchiveException class, 13, 145
- CArray classes, 110–112
- CAssoc structure, 124, 126
- CAsyncSocket, 28
- CATCH macro, 144
- CATCH_ALL macro, 144
- CBEdit class, 28
- CBitmap class, 20, 62, 250
- CBitmap Detach() function, 255
- CBitmapButton class, 19, 250
- CBlockHeader structure, 191–192
- CBrush class, 20, 62
- CButton class, 18–19
- CBYTEArray class, 14, 108
- CB_ADDSTRING message, 214
- CCheckList::PreDrawItem() member function, 254–255
- CCheckListBox class, 19, 250, 252–255
- CCheckListBox..SetCheck() function, 253–254
- CCClientDC class, 20, 60
- CCmdTarget class, 16, 25, 65, 70, 263, 467, 471, 602–604, 611–613
 implementing IUnknown functions, 473–474
 type libraries, 617
- CCmdTarget EnableAutomation() function, 603–604
- CCmdTarget::EnableTypeLib() function, 617
- CCmdTarget FromIDispatch() function, 604
- CCmdTarget::GetDispatchIID() function, 617
- CCmdTarget GetIDispatch() function, 604
- CCmdTarget::GetInterface() function, 484–485
- CCmdTarget .GetStandardProperty() function, 613–614
- CCmdTarget::GetTypeInfoCount() function, 617
- CCmdTarget::GetTypeInfoOfGuid() function, 617
- CCmdTarget GetTypeLib() function, 617
- CCmdTarget::GetTypeLibCache() function, 617
- CCmdTarget InternalQueryInterface() function, 473
- CCmdTarget OnCmdMsg() function, 82–83
- CCmdTarget SetStandardProperty() function, 613–614
- CCmdUI class, 16
- CCColorDialog class, 18, 226, 228, 236
- CCComboBox class, 19, 250
- CCCommandLineInfo class, 17, 41–42
- CCCommonDialog class, 228–229, 235
- CCCommonDialog OnOK() function, 228–229
- CCControlBar class, 23, 365–392
 data members, 369–370
 implementation, 368–369
 layout management, 390–391
 member functions, 370–371
- CCControlBar EnableDocking() function, 371–372
- CCControlBar GetBarInfo() function, 387–388
- CCControlBar OnLButtonDown() function, 381–382
- CCControlBarInfo class, 385–387
- CCreateContext structure, 276–277, 290
- CCriticalSection class, 17
- CCtrlView class, 327–328
- CDaoDatabase class, 27
- CDaoException class, 27, 145
- CDaoFieldExchange class, 28
- CDaoQueryDef class, 27
- CDaoRecordset class, 27
- CDaoRecordView class, 27
- CDaoWorkspace class, 27
- CDatabase class, 27
- CDataExchange class, 19, 217–219

- CDataExchange**::Fail() function, 221–222, 224–225
CDBException class, 13, 145
CDC class, 60–61
CDC::Detach() function, 60
CDC::StartDoc() function, 306
CDC.StartPage() function, 306
CDialog class, 18, 201–202, 204–207, 242
 control initialization, 212–215
 exchange and validation, 215–225
 modal dialogs, 203, 207–210
 modeless dialogs, 210–211
 termination, 212
CDialog::Create() member function, 210–211
CDialog::CreateIndirect() function, 211
CDialog::DoDataExchange() function, 222
CDialog.DoModal() member function, 208–210, 240
CDialog::EndDialog() function, 212
CDialog.HandleInitDialog() function, 212–213
CDialog:OnInitDialog() function, 273
CDialog:OnNotify() member function, 233–234
CDialog::OnOK() function, 222, 225
CDialog::PreModal() function, 231
CDialogBar class, 366
CDialogBar.Create() function, 57
CDocItem class, 24, 550–551
CDockBar class, 376–378
CDockBar.Create() function, 57
CDockBar.DockControlBar() function, 375–376
CDockState class, 383–385
CDockState.SaveState() function, 388–389
CDocManager class, 267–271, 289–290
CDocManager.OnFileNew() member function, 270–271, 291
CDocManager.OnFileOpen() function, 291
CDocManager::OpenDocumentFile() function, 291
CDocTemplate class, 21, 265–266, 273–276, 284, 290
CDocTemplate::AddDocument() function, 282
CDocTemplate::CreateNewDocument() function, 274–275, 280, 290, 292
CDocTemplate::CreateNewFrame() function, 274–276, 280, 290, 292
CDocTemplate::GetDocString() function, 273
CDocTemplate::InitialUpdateFrame() function, 279
CDocTemplate::OpenDocumentFile() function, 271, 291–293
CDocument class, 21, 70, 262–263, 282–296
CDocument::GetDocTemplate() function, 282
CDocument::GetFile() function, 285, 295
CDocument.OnFileSave() function, 283
CDocument::OnNewDocument() function, 284
CDocument::OnOpenDocument() function, 279, 284–285
CDocument::OnSaveDocument() function, 171–172, 286–287
CDocument::OnUpdateFileSendMail() member function, 28
CDocument::Serialize() function, 285
CDocument::SetPathName() function, 279
CDocument.UpdateAllViews() function, 287–288
CDragListBox class, 19, 250, 256
CDumpContext class, 13, 181–183
CDumpContext.OutputString() function, 183
CDWordArray class, 14, 108
CDynLinkLibrary helper class, 410–413
CDynLinkLibrary::CDynLinkLibrary() function, 411–412
CEdit class, 18–19, 250
CEditView class, 23, 327, 330
CEvent class, 17
CException class, 13, 145–148
CFieldExchange class, 27
CFile class, 15, 129–133, 142, 148
CFile.Close() function, 299
CFile::GetBufferPtr() function, 140
CFile.GetStatus() function, 298
CFile.Open() function, 130, 285, 298
CFileDialog class, 18, 226, 228–234, 236
CFileDialog::DoModal() function, 230–231
CFileDialog::GetFileName(), 234–235
CFileException class, 13, 145–148
CFindReplaceDialog class, 18, 226, 228
CFont class, 20, 62
CFontDialog class, 18, 226, 228, 236
CFontDlg class, 204
CFontHolder class, 26, 627
CFormView class, 23, 324–327
CFormView::Create() function, 325–327
CFrameWnd class, 18, 265, 279–281, 290
CFrameWnd.CreateView() function, 280
CFrameWnd.DockControlBar() function, 374
CFrameWnd.EnableDocking() function, 372–373
CFrameWnd.FloatControlBar() function, 379
CFrameWnd.GetDockControlState() function, 382–383
CFrameWnd.InitialUpdateFrame() function, 279, 281
CFrameWnd.LoadFrame() function, 57, 280
CFrameWnd::OnCmdMsg() function, 81–82
CFrameWnd::OnCreate() function, 280–290
CFrameWnd::OnInitialUpdate() function, 279
CFrameWnd::PreCreateWindow() function, 57
CFrameWnd.RecalcLayout() function, 390–391
CFrameWnd.SaveBarState() function, 382
CFrameWnd::SetActiveView() function, 279
CF_BITMAP format, 505
CF_EMBEDDSOURCE format, 505

- CF_LINKSOURCE format, 505
 CF_TEXT format, 505
CGdiObject class, 20, 62
CGenericApp class, 37
`_ChangePropPageFont()` function, 248–249
CHandleMap class, 44–46
CHandleMap `FromHandle()` function, 46
 Character strings, 94
CHeaderCtrl class, 19, 256
`Checkpoint()` member function, 178
CEdit class, 28
 Child window, 18
CHotKeyCtrl class, 19, 256
CImageList class, 19, 256
 Class composition, 474
 Class factories, 25–26, 447–448
 adding to applications, 494–495
 CoMath class, 462–464
 exporting from DLL (Dynamic Link Library), 497–498
 in-proc server, 449
 MFC (Microsoft Foundation Classes) support, 487–498
 out-of-proc server, 449–450
 registration, 489–491
 Class virtual table, 198
 Class Wizard, 154, 201
`DoDataExchange()` member function, 216
 Classes, 5
 adding serialization, 162–163
 application framework, 12, 21–22
 C++, 2, 436
 COM (Component Object Model), 433–438
 common dialogs, 226–227
 control, 250–257
 declaration subsections, 656
 deriving, 2, 5–6
 enhanced user interface, 333–396
 exception-handling, 13–14
 file, 15
 general-purpose, 12–15
 grouping into modules, 3
 information about, 13
 inheritance, 5–6
 keeping tabs on, 284
 member functions, 5
 message-handling procedures, 38
 nested, 458–462, 465–466
 OLE common dialog declarations, 237–238
 OLE controls, 624–628
 OLE drag-and-drop, 531–534
 passing name as argument, 155
 printing out state of, 175
 retrieving message map, 74
 root, 13
 selecting proper, 2
 sharing functions, 3
 singly rooted hierarchy, 151
 static members, 155
 synchronization object, 17
 testing type, 154
 time and time span, 15
 utility, 93–148
 vs structs, 5
 Windows API, 12, 16–20
 windows derived from, 18
 zero-initializing, 207
 ClassWizard
 automation, 596
 dispatch maps, 609–610
CLASS_E_CLASSNOTAVAILABLE error code, 498
CList class, 120–121
CListBox class, 18–19, 250
CListBox `AddString()` function, 252
CListBox `SetItemDataPtr()` function, 273
CListCtrl class, 19, 256
CListView class, 23, 330
CLongBinary class, 27
Close() function, 299
CloseClipboard() function, 502–503
CMap class, 126
CMap `NewAssoc()` function, 128
CMapPtrToPtr class, 14, 77, 108
CMapPtrToWord class, 14
CMapStringToOb class, 15, 108
CMapStringToPtr class, 15, 108
CMapStringToString class, 15, 108, 122–123
CMapWordToOb class, 15, 108
CMapWordToPointer class, 15, 124
CMapWordToPtr class, 108, 123–124
CMapWordToPtr `GetAssocAt()` member function, 128
CMapWordToPtr `GetNextAssoc()` member function, 129
CMapWordToPtr `:operator[]`, 127
CMDIChildWnd class, 18, 265
CMDIFrameWindow `.PreCreateWindow()` function, 57
CMDIFrameWnd class, 18
CMD_GETSPEC message, 235
CMemFile class, 129, 136–140, 142
CMemFile `.GetStatus()` member function, 137
CMemFile `.GrowFile()` member function, 138–139
CMemFile `.Read()` member function, 139
CMemFile `.Seek()` member function, 146–148
CMemoryException class, 13, 145, 148
CMemoryState class, 14, 178–179, 187–189, 192–194
CMemoryState `.Checkpoint()` function, 193
CMemoryState `.Difference()` function, 193–194
CMemoryState `.DumpAllObjectsSince()` function, 194
CMemoryState `:DumpStatistics()`, 194
CMenu class, 20
CMetaFile `.Create()` function, 61
CMetaFileDC class, 20, 60
CMiniDockFrameWnd class, 372, 379–380
CMiniDockFrameWnd `.Create()` function, 380–381
CMiniFrameWnd class, 392–393
CMirrorFile class, 296–299
CMirrorFile `.Close()` function, 298–299
CMirrorFile `.Open()` function, 296–298
CModalDialog class, 202
CMultiDocTemplate class, 21, 160, 266, 273, 277–278
CMultiDocTemplate `.AddDocument()` function, 278

- CMultiDocTemplate::OpenDocumentFile()**
 function, 278–279, 292
- CMultiLock class**, 17
- CMutex class**, 17
- CMyClass class**, 155, 170–171
- CMyClass..AssertValid() member function**, 176
- CMyClass::CreateObject() function**, 161, 196–197
- CMyClass::GetRuntimeClass() function**, 155
- CMyClass::Serialize() function**, 171–172
- CMyDoc .OnOpenDocument() function**, 292
- CMyDocument .OnNewDocument() function**, 293
- CMyView .OnDraw() function**, 307
- CMyView::OnFilePrint() function**, 304
- CMyView::OnPreparePrinting() function**, 303
- CNewType dialog**, 270–271
- CNewType::OnOK() function**, 273
- CNewTypeDlg class**, 271–273
- CNewTypeDlg OnInitDialog() function**, 272–273
- CNode structure**, 116
- CNotSupportedException class**, 13, 145, 148
- COBArray class**, 14, 108
- CoBetterMath object**, 469–470
- CObject class**, 13, 151–162, 198–199, 263
 diagnostic support, 174–195
 dynamic creation, 159–162
 features, 151–153
 memory diagnostics, 177–179
 operator, 153
 RTCI (run-time class information), 153–159
 run-time error checking, 175–176
 serialization, 162–163, 169–170
 tracing read and write of derived object, 170–172
- CObject..AssertValid() function**, 185
- CObject::Dump() function**, 184
- CObject::IsKindOf() function**, 158–159
- CObject. IsSerializable() function**, 170
- CObject..Serialize() function**, 162–163, 170
- COBList class**, 14, 108
- COccManager class**, 629
- CoCreateInstance() function**, 449, 453
- CoFreeUnusedLibraries() function**, 451
- CoGetClassObject() function**, 449, 453
- CoGetMalloc() function**, 443–444
- COleBusyDialog class**, 237
- COleChangeIconDialog class**, 237
- COleChangeSourceDialog class**, 237
- COleClientItem class**, 24, 552–554, 562–564
- COleClientItem::Close() function**, 575
- COleConnectionPoint class**, 26, 627
- COleConnPtContainer class**, 636
- COleControl class**, 26, 625–627
- COleControl .FireEvent() function**, 640
- COleControlContainer class**, 628
- COleControlModule class**, 26, 625
- COleControlSite class**, 628–629
- COleConvertDialog class**, 237
- COleDataObject class**, 26, 511, 513, 523–525
- COleDataObject..Attach() function**, 524
- COleDataObject::AttachClipboard() function**, 513, 524–525
- COleDataObject::BeginEnumFormats() function**, 525
- COleDataObject::Detach() function**, 524
- COleDataObject..EnsureClipboardObject() function**, 524–525
- COleDataObject::EnumFormatEtc() function**, 525
- COleDataObject::GetData() function**, 525
- COleDataObject. GetFileData() function**, 525
- COleDataObject::GetGlobalData() function**, 525
- COleDataObject::GetNextFormat() function**, 525
- COleDataObject..IsDataAvailable() function**, 513, 525
- COleDataSource class**, 25, 511–513, 515–523, 531
- COleDataSource.:CacheData() function**, 513, 519–520
- COleDataSource .CacheGlobalData() function**, 513, 519–520
- COleDataSource::DAdvise() function**, 523
- COleDataSource::DoDragDrop() function**, 528–529
- COleDataSource::DUnadvise() function**, 523
- COleDataSource::EnumDAdvise() function**, 523
- COleDataSource::EnumFormatEtc() function**, 522–523
- COleDataSource::GetCanonicalFormatEtc() function**, 522
- COleDataSource..GetData() function**, 514
- COleDataSource::GetDataHere() function**, 522
- COleDataSource :GetFileData() function**, 514
- COleDataSource::GetGlobalData() function**, 514
- COleDataSource::QueryGetData() function**, 522
- COleDataSource :SetClipboard() function**, 512
- COleDataSource::XDataObject GetData() function**, 521
- COleDialog class**, 237–239
- COleDispatchException class**, 145
- COleDispatchImpl Invoke() function**, 613
- COleDocument class**, 24, 550–551
- COleDropSource class**, 26, 532–533
- COleDropTarget class**, 26, 529, 533–534
- COleException class**, 13, 145
- COleInsertDialog class**, 237
- COleInsertDlg CreateItem() function**, 564–567
- COleIPFrameWnd class**, 25
- COleLinkingDoc class**, 24, 554–557
- COleLinksDialog class**, 237
- COleObjectFactory class**, 25, 487–491, 495–497

- COleObjectFactory.** OnCreateObject() function, 493
- COleObjectFactory.**:Register() function, 490
- COleObjectFactory.**:Revoke() function, 490
- COleObjectFactory**: UpdateRegistry() function, 490
- COleObjectFactoryEx** class, 26, 489
- COlePasteSpecialDialog** class, 237–238
- COlePropertiesDialog** class, 237, 239
- COlePropertyPage** class, 26, 239, 627, 651–652
- COleResizeBar** class, 366
- COleResizeBar.** Create() function, 57
- COleServerDoc** class, 25, 557–561
- COleServerDoc.** SaveEmbedding() function, 576–577
- COleServerItem** class, 24, 561
- COleStreamFile** class, 130
- COleTemplateServer** class, 25
- COleUILinkInfo** class, 239
- COleUpdateDialog** class, 237
- Collection** classes, 14–15, 107–129
 - arrays, 108–114
 - lists, 108, 114–122
 - maps, 14–15, 108, 122–129
 - nontemplate, 109
 - template, 109
- Color**, 18
- Columnar list boxes**, 19
- COM** (Component Object Model)
 - aggregation, 468–473
 - allocating objects, 446–450
 - automation, 584–602
 - binary reuse, 468
 - call/use/release rules for using objects, 443–444
 - class factories, 446–450
 - class registration, 452–453
 - GUIDs (Globally Unique Identifiers), 439
 - interfaces, 263, 437–440, 485–487, 585–587
 - object lifetime management, 440–443
 - object servers, 444–445
- COM (Component Object Model)** classes
 - defining, 435–437
 - inheritance, 482–484
 - instance creation, 453
 - making aggregatable, 471–472
 - MFC (Microsoft Foundation Classes), 464–467
 - multiple inheritance, 438
 - multiple interfaces, 454–464
- CoMath** class, 455, 586
 - class factory, 462–464
 - header file, 455–459, 479–482, 462
 - interface map, 484–485
 - MFC (Microsoft Foundation Classes), 467, 479–482
 - multiple inheritance, 455–458
 - nested classes, 458–462
 - new instance, 493
 - object, 468–470
 - ODL script describing, 598
- Combo box**, 19
- Command line parsing**, 17
- CommandLineInfo** class, 41
- Commands**
 - messages as, 75, 81–85
 - paths, 85–87
- Common #defines**, 659–660
- Common Color dialog box**, 202
- Common dialogs**, 18, 225–239
 - classes, 226–227
 - example listing, 226–227
 - OLE, 236–239
 - where to look for, 228
 - Windows, 227
- Compound documents**, 24–25
- Compound files**, 546–547
- Computer-based training applications**
 - hooks, 57–58, 76
- Concatenation (###) operator**, 155
- Connection maps**, 637–639
- CONNECTION_ID** macro, 638
- CONNECTION_PART** macro, 638
- Containers**
 - embedded or linked OLE item, 24
 - exchanging properties with controls, 26
 - OLE controls, 628–629
- Content objects**, 547
- Context-sensitive help**, 22
- Contracts**, 2
- Control bars**, 23–24, 55, 365–382
 - dialog bars, 24
 - docking, 368–376
 - dragging, 381–382
 - floating implementation, 378–381
 - layout management, 390–391
 - operation, 366–368
 - persistence, 382–389
 - status bars, 24
 - toolbars, 23–24
- Control classes**, 250–257
- Control views**, 23
- Controls**, 19–20
 - command messages from, 263
 - exchanging properties with containers, 26
 - Explorer-like tree, 20
 - generalized notification, 89–90
 - initialization, 212–215
 - limitations, 22
 - messages, 81
 - modifying properties, 26
 - OLE (Object Linking and Embedding), 26
 - wiring data member to, 215–216
- CopyBeforeWrite()** member function, 105
- CoRegisterClassObject()** function, 449–450
- CoSomeObject** COM class, 441, 448
- CPageSetupDialog** class, 226, 228
- CPageSetupDialog:** PaintHookProc() procedure, 236
- CPaintDC** class, 20, 60–61
- CPalette** class, 20, 62
- CPen** class, 20, 62
- CPictureHolder** class, 26, 627
- CPlex** structure, 116–119, 124
- CPlex::Create()** function, 118–119
- CPlex::FreeDataChain()** member function, 120–121
- CPoint** class, 15, 106
- CPP** files, 154
- CPreviewView** class, 311–317

CPreviewView::OnDraw() member function, 315–316
CPreviewView::SetPrintView() member function, 314–315
CPrintDialog class, 18, 226, 228
CPrintInfo structure, 300–303
CPrintInfo() function, 302
CPrintingDialog structure, 304
CPrintPreviewState structure, 309
CProgressCtrl class, 19, 256
CPropertyPage class, 19, 240, 242–243, 249–250
CPropertyPage::SetModified() function, 241
CPropertySheet class, 19, 240, 242–246, 249–250
CPropertySheet::AddPage() function, 240, 247–248
CPropertySheet::BuildPropPageArray() function, 248
CPropertySheet::DoModal() function, 246–247
CPropExchange class, 26, 628
CPrtList class, 14
CPtrArray class, 14
CPtrList class, 108
Create() function, 42, 251
CreateDialog() API function, 204
CreateDialogIndirect() API function, 204
CreateInstance() function, 447
CreateObject() method, 173
CreateThread() API function, 58
CreateWindowEx() API function, 76
CRecentFileList class, 394–396
CRecordset class, 27
CRecordView class, 27
CRect class, 15, 106–107
CRect::DeflateRect() member function listing, 107
CResourceException class, 14, 145
CRgn class, 20, 63
CRichEditCtrl class, 20, 256
CRichEditView class, 23, 330–331
CRuntimeClass structure, 13, 51, 155–156, 162, 166, 174, 196–197
CRuntimeClass::CreateObject() function, 197, 275–276
CRuntimeClass..CreateObject() function listing, 161
CRuntimeClass::IsDerivedFrom() function, 159
CScrollBar class, 19, 251
CScrollView class, 22, 317–324
CScrollView::OnPrepareDC() function, 322–323
CScrollView::SetScrollSizes() function, 320–322
CSemaphore class, 17
CSharedFile class, 130, 141–143
CSimpleException class, 148
CSingleDocTemplate class, 21, 266, 273, 277–278
CSingleDocTemplate::OpenDocumentFile() function, 292
CSingleLock class, 17
CSize class, 15, 106
CSliderCtrl class, 20, 256
CSocket class, 28
CSpinButtonCtrl class, 20, 256
CSplitterWnd class, 23, 333–365
 cosmetic data members, 340
 creation/layout data members, 340
 declaration, 337–339
 drawing, 352–357
 drawing member functions, 342
 encapsulated data types, 339–340
 general member functions, 341
 hit test member functions, 342
 hit testing, 358–360
 initialization, 343–349
 layout member functions, 341–342
 message handlers, 342–343
 pane management, 349–352
 tracking data members, 341
 tracking member functions, 342
CSplitterWnd::Create() function, 343–344
CSplitterWnd::CreateCommon() function, 57, 346–347
CSplitterWnd::CreateStatic() function, 344–345
CSplitterWnd::CreateView() function, 347–349
CSplitterWnd::HitTest() function, 359–360
CSplitterWnd..OnDrawSplitter() function, 353–355
CSplitterWnd::OnInvertTracker() function, 362
CSplitterWnd::OnLButtonDown() function, 361–362
CSplitterWnd::OnPaint() function, 355–357
CSplitterWnd::SplitColumn() function, 349–352
CStatic class, 19, 251
CStatusBar class, 366
CStatusBarCtrl class, 20, 256
CStdioFile class, 15, 129, 133–135, 142
CStdioFile::ReadString(), 135
CString class, 15, 94–106
 assignment operator, 103–104
 calling _t string routines, 105–106
 collection utility functions, 118
 constructor, 102
 CStringData helper structure, 97–100
 declaration, 96–97
 example listings, 94–95
 implementation, 96–100
 member functions, 95
 operators, 94
 reference counting, 96, 98–105
 referring to locked data, 103
CString::AllocBuffer() function, 98
CString::CopyBeforeWrite() member function, 104
CString::Find() member function, 105–106
CString::LockBuffer() function, 103
CString::MakeUpper() member function, 104
CString::Release() member function, 105
CStringArray class, 14, 108
CStringData helper structure, 97–100
CStringData::nRefs reference count, 101
CStringList class, 14, 108, 115–118
CStringList::GetNext() member function, 122
CStringList::NewNode() member function, 117–118
CStringList::RemoveAll() member function, 119–120
CSyncObject class, 17

- CTabCtrl class, 20, 242, 256
 CTestView class, 72
 CTime class, 15, 106
 CTimeSpan class, 15, 106
 CToolbar class, 24, 366
 CToolBarCtrl class, 20, 256
 CToolTipCtrl class, 20, 256
`_CrDumpMemoryLeaks()` function, 195
 CTreeCtrl class, 20, 256
 CTreeCtrl::InsertItem() member function, 257
 CTreeView class, 23, 329–330
 CTreeView::CTreeView() function, 329
 CTreeView GetTreeCtrl() function, 330
 CTreeView PreCreateWindow() function, 330
 CtypeLibCache class, 616
 CUintArray class, 14, 108
 CUserException class, 14, 145
 Custom interface, 454–455
 Custom verbs, 491–493
 CVBControl class, 19
 CView class, 21–22, 263–264, 279, 288–311, 529, 534
 data members, 288
 message map entry, 302
 CView::Create() function, 57
 CView::DoPreparePrinting() function, 303
 CView::DoPrintPreview() function, 309–311
 CView::GetDocument() function, 288
 CView::GetParentFrame() function, 287, 290
 CView::OnActivateView() function, 281
 CView::OnBeginPrinting() function, 303
 CView::OnCmdMsg() function, 82
 CView::OnDraw() function, 306
 CView::OnFilePrint() member function, 302–306
 CView::OnFilePrintPreview() function, 308–309
 CView::OnInitialUpdate() member function, 279, 281, 283
 CView::OnPaint() function, 288–289
 CView::OnPrint() function, 300, 306
 CView::OnUpdate() function, 288
 CWaitCursor class, 17
 CWinApp class, 16–17, 39–42, 52–53, 70, 266–267, 289, 425
 member functions, 41–42, 269
 profile strings, 53
 variables, 41–42
 vs CDocManager class, 267–271
 CWinApp::AddDocTemplate() function, 266, 269
 CWinApp::CloseAllDocuments() member function, 269
 CWinApp::DoPromptFileName() function, 269, 286
 CWinApp::InitInstance() function, 42, 266, 289–290
 CWinApp::GetFirstDocTemplatePosition() member function, 269–270
 CWinApp::GetNextDocTemplate() member function, 269–270
 CWinApp::GetOpenDocumentCount() member function, 269
 CWinApp::OnFileNew() function, 269, 291
 CWinApp::OnFileOpen() function, 269, 291
 CWinApp::OpenDocumentFile() function, 269, 291
 CWinApp::PreTranslateMessage() function, 91–92
 CWinApp::RegisterShellFileType() member function, 269
 CWinApp::Run() function, 42, 54, 58, 91
 CWinApp::SaveAllModified() member function, 269
 CWindowsDC class, 20, 60
 CWinThread class, 16–17, 58, 419–424
 CWinThread::CreateThread() function, 420–423
 CWinThread::OnIdle() function, 427–428
 CWinThread::Run() function, 426–429
 CWnd class, 16, 18, 29, 38, 42–44, 70, 77, 263, 529
 CWnd::Attach() function, 46
 CWnd::CenterWindow() function, 246
 CWnd::Create() function, 56, 280
 CWnd::CreateDlgIndirect() function, 209
 CWnd::CreateEx() function, 76
 CWnd::Default() function, 44
 CWnd::DefWindowProc() function, 44, 78, 83
 CWnd::DestroyWindow() function, 203
 CWnd::Detach() function, 46
 CWnd::EndModalLoop() function, 210
 CWnd::ExecuteDlgInit() function, 213–214, 222
 CWnd::FromHandle() function, 45–46
 CWnd::FromHandlePermanent() function, 78
 CWnd::GetActiveView() function, 82
 CWnd::GetActiveWindow() function, 45
 CWnd::OnCmdMsg() function, 82
 CWnd::OnCommand() function, 81
 CWnd::OnNcDestroy() function, 45
 CWnd::OnNotify() member function, 89
 CWnd::OnWndMsg() function, 78–81, 89
 CWnd::PreTranslateMessage() function, 91–92
 CWnd::RunModalLoop() function, 209–210, 222, 247
 CWnd::SetParent() API function, 376
 CWnd::UpdateData() function, 222–224
 CWnd::WalkPreTranslateTree() function, 91–92
 CWnd::WindowProc() function, 78
 CWordArray class, 14, 108–110
 CWordArray::InsertAt() function, 113
 CWordArray::SetSize() function, 111
 DAO (Data Access Objects), 27–28
 Data
 handling, 21
 multiple views, 21–22
 representing on-screen, 260–262
 separating from GUI code, 260
 viewing on-screen, 21–22, 263–264
 Data consumers, 506
 Data management, 259–262
 Data members, 110

- Data producers, 506
 Data sets, 262
 Data source, 262
 Data structures and message maps, 70–71
 Data transfers, 25–26 *See also* Transferring data
 Data-access processing exceptions, 13
DBCS (Double Byte Character Convention), 133
DDE (Dynamic Data Exchange), 501, 503
DDV_MaxChars() routine, 221–222
DDX/DDV (dynamic data exchange/dialog data validation), 18–19, 215–225
 adding DDV control validation listing, 216
 operation, 217–224
 routines, 219–222
DDX_Text() routine, 219–220
Debugger, 197
 retaining record of last message, 78
DEBUG_NEW macro, 189
DECLARE macros, 153–154
DECLARE_CONNECTION_MAP macro, 638
DECLARE_DYNACREATE() macro, 493
DECLARE_DYNAMIC macro, 154–155, 160, 414–415
DECLARE_DYNCREATE macro, 154, 160, 196
DECLARE_EVENTSINK_MAP macro, 642, 644
DECLARE_INTERFACE_MAP() macro, 477–478, 495, 637
DECLARE_MESSAGE_MAP macro, 71–72
DECLARE_OLECREATE() macro, 494–495
DECLARE_OLETYPELIB macro, 615
DECLARE_SERIAL macro, 154, 162, 167–168
DefWindowProc() function, 38, 44, 56, 65, 76–77
 Delayed rendering, 515–516
DelayRenderData() function, 521
DeleteMetaFile() function, 61
 Derived classes, 13
 safeness of polymorphic cast, 154
 Device contexts, 20, 59–61
 customizing, 322–323
 encapsulating, 20
 handles (HDC), 60, 263
 minimizing usage, 321
 Device drivers, 59
 Device independence, 59
 Diagnostic functions, 13
 Diagnostic output, 174–175, 182–187
 Dialog bars, 24
 Dialog boxes, 18–19 *See also* Dialogs
DialogBox() API function, 204
DialogBoxIndirect() API function, 204
 Dialogs
 auto-centering feature, 78
 color selection, 18
 command path to, 87
 common, 18, 225–239
 control initialization, 212–215
 control limitations, 22
DDX/DDV (dynamic data exchange/dialog data validation), 18–19, 215–225
 displaying, 201
 font selection, 18
 modal, 201–203
 modeless, 201–204
 operations, 204
 OLE controls, 622, 630–631
 printing, 18
 property sheets, 19
 searching and replacing text, 18
 selecting files from directory, 18
 tabbed, 19, 239–250
 user choices, 271–273
 Dictionaries, 108 *See also* Maps
 Dispatch maps, 25, 604–606, 611–613
 ClassWizard, 609–610
 DISPIDs, 610
 expanded by preprocessor, 610
 expanding macros, 609
 implementing, 606–607
DispatchCmdMsg() function, 83
DispatchMessage() function, 37
DISP_FUNCTION macro, 607
DISP_PROPERTY macro, 607–608
DISP_PROPERTY_EX macro, 608–609
DISP_PROPERTY_NOTIFY macro, 608
DISP_PROPERTY_PRAM macro, 609
DLGINIT resource, 213–214
DLL (Dynamic Link Library)
 AFXDLLs, 407–408
 base class for holding OLE controls, 26
 exporting class factories, 497–498
 extension initialization and cleanup, 414
 macros, 414–417
 resources, 408–414
 states, 399
 under Windows, 445
 unloading, 450–451
 USRDLLs, 407–408
DllCanUnloadNow() function, 451, 498
DllGetClassObject() function, 449, 497–498, 632
DOCINFO structure, 303
 Document templates, 21, 265–273, 289–290
 Document/view architecture, 21–22, 259–262
 advanced, 295–331
 document creation, 291–294
 document templates, 265–273
 document/view frames, 264–265
 documents, 262–263
 frame window, 264–265
 interdependencies, 289–290
 keeping tabs on classes, 284
 open documents, 277–279, 291
 storing elements, 276–279
 tying together document/view/frame, 273–276
 views, 263–264
 Documents, 261–262
 binding user interface directly to, 263
 creation, 284–285

- linked and embedded OLE items, 24
 list of open, 277–279
 new, 274–275, 279–291
 opening, 278–279, 291
 programmatic commands, 263
 saving, 286–287
 views, 21
- DoDataExchange()** member function, 215–216
- DoDragDrop()** function, 527
- DOModal()** function, 203, 226
- DoPreparePrinting()** function, 300
- DoPrintPreview()** function, 309–311
- Double word values, 14
- Doubly-linked lists, 108
- Dragging and dropping
 data transfers, 26
 items into list boxes, 19
- Drawing graphics, 60–61
- Drawing objects, 20
- Drawing tools, 62–63
- Dump of all objects, 179
- Dump() member function, 175
- DumpAllObjectsSince() member function, 179
- DumpStatistics() member function, 179
- Duntemann, Jeff, 435
- Dynamic creation, 13, 159–162
- Dynamic splitter windows, 23, 333–336
- Dynamic strings, 15
- Early binding, 587
- Edit controls, 19, 23
- Elapsed time, 15
- Employee class, 6
- EmptyClipboard() function, 502
- EnableMenuItem() function, 16
- Encapsulation, 1–2, 4–5, 15
- EndDoc() function, 306
- EndPaint() function, 20
- END_CONNECTION_PART** macro, 637–638
- END_EVENTSINK_MAP** macro, 642, 644
- END_INTERFACE_MAP()** macro, 478–479, 637
- END_INTERFACE_PART()** macro, 476, 495, 637
- END_MESSAGE_MAP** macro, 71–74, 87
- END_TRY** macro, 144
- Enhanced user interface classes, 333–396
- Enhanced views, 22–23
- Event sink, 642–646
- Event sink maps, 644–646
- Events
 firing, 640
 MFC (Microsoft Foundation Classes), 641–642
 OLE controls, 639–641
- Exception handling
 classes, 13–14, 143–148
 macros, 143
 notifying user that exception occurred, 146–148
 retaining record of last message, 78
- Exceptions, 13–14
- EXE files, 445
- ExitInstance() function, 42
- Explorer-like tree control, 20, 23
- ExternalAddRef() function, 467, 471, 473–474
- ExternalQueryInterface() function, 467, 471, 473–474
- ExternalRelease() function, 467, 471, 473–474
- File classes, 15
- File handles, 131–132
- FILE pointer, 134
- File utility classes, 129–143
- File-operation exceptions, 13
- Files
 accessing, 129
 binary I/O, 15
 buffered I/O, 15, 129, 133–135
 closing, 298–299
 encapsulating I/O, 15
 implementing shared memory, 129–130
 memory, 136–138
 modes, 130
 name manipulation, 132–133
 new, 298
 reasons for exceptions, 146
- saving original and writing to temporary, 299
- selecting from directory, 18
- truncating existing, 298
- FileNew** command, 266
- FileOpen** command, 266
- FireCurrentPriceChanged()** function, 640
- Font objects, 26
- Fonts, 18, 62
- Form views, 22
- FORMATEC structure, 504–506, 508
- Frame windows, 18, 25, 85, 264
- Frames, 290
 active view, 82
 new, 274–280
 updating views, 281
- Frameworks, 7
 initializing, 52–53
- FromHandle()** member function, 60
- Function signatures, 83–85
- Functions
 checking for exceptions, 143–148
 signature, 643
- Gamma, Erich, 59
- GDI (Graphics Device Interface)
 device contexts, 59–61
 graphic objects, 61–63
 objects, 62
 support, 20, 58–61
- General-purpose classes, 12–15
- GetActiveWindow()** API function, 45
- GetBufferPut()** member function, 140
- GetClassID()** function, 454
- GetClipboardData()** function, 503
- GetDiskFreeSpace()** function, 298
- GetFile()** function, 295–296
- GetFirstDocPosition()** function, 265
- GetFirstViewPosition()** function, 288
- GetIDsOfNames()** function, 611–612
- GetMessage()** function, 37
- GetMessage()–DispatchMessage()** loop, 39, 42, 58
- GetMessageMap()** function, 72, 74
- GetMinPage()** function, 302
- GetModuleFilename()** function, 53
- GetNextDocPosition()** function, 265
- GetOpenFile()** API function, 235

- GetPropCheck() function, 653
 GetSaveFile() API function, 235
 GetStatus() function, 298
 GetTabControl() member function, 249
 Global objects, 52
 Granularity, 660–661
 Graphic modes, 60
 Graphic objects, 60–63
 Graphical lists, 19
 Graphics
 drawing, 60–61
 drawing tools, 62–63
 painting, 61
 GUIDs (Globally Unique Identifiers), 439
- Handle maps, 45
 Hashing, 125–129
 HashKey() member function, 126, 128
 HCBT_CREATEWND code, 76–77
 Header files, 6, 154
 Helm, Richard, 59
 Help, 22
 Help for Current Task (F1) key, 22
 Help for Next Item (Shift-F1) key combination, 22
 Help menu, 22
 High-level abstractions, 12, 22–24, 28
 High-level architecture support, 10–11
 HitTestValue enumeration declaration, 358–359
 Hooking into message loop, 91, 92
 Hungarian notation, 657
- I/O (input/output), 15
 IAddSubtract interface, 468–470
 IAdviseSink interface, 548, 552–553
 IClassFactory interface, 447–448
 IClassFactory::CreateInstance() function, 453, 493
 IClassFactory::LockServer() function, 451, 493
 IConnectionPoint interface, 635
 IConnectionPointContainer interface, 635
 IDataObject interface, 25, 438, 507–509, 549, 559–560
- classes, 511–526
 OLE Clipboard, 509–510
 IDataObject::DAdvise() function, 509
 IDataObject::DUadvise() function, 509
 IDataObject::EnumDAdvise() function, 509
 IDataObject::EnumFormatEtc() function, 509
 IDataObject::GetCanonicalFormatEtc() function, 508
 IDataObject::GetData() function, 508, 521
 IDataObject::GetDataHere() function, 508
 IDataObject::QueryGetData() function, 508
 IDataObject::SetData() function, 508
 IDispatch interface, 438, 584, 587–596
 manually implementing, 591–596
 type information, 599–601
 IDispatch::CreateStdDispatch() function, 601
 IDispatch::GetIDsOfNames() function, 591, 595–596
 IDispatch::GetTypeInfo() function, 600
 IDispatch::GetTypeInfoCount() function, 600
 IDispatch::Invoke() function, 589–595
 IDropSource interface, 526
 IDropTarget interface, 438, 526
 IDSPIDs, 610
 ILockBytes interface, 545–547
 Images, 19
 IMalloc interface, 443–444, 585
 IMath interface, 454–455, 458, 585–586
 IMPLEMENT macro, 153–154
 IMPLEMENT_DYNAMIC macro, 154, 156–158, 415–416
 IMPLEMENT_DYNCREATE macro, 154, 160–161, 196, 493
 IMPLEMENT_OLECREATE() macro, 494–495
 IMPLEMENT_OLETYPEDLIB macro, 615
 _IMPLEMENT_RUNTIMECLASS utility macro, 161, 168
- IMPLEMENT_SERIAL macro, 154, 162, 167–168, 173, 196
 IMultiplyDivide interface, 468–470
 In-place items, 547–548
 In-proc server, 444–445, 449–451
 Inheritance, 1–6
 C++, 2–3
 COM (Component Object Model)
 classes, 482–484
 message maps, 74
 multiple, 455–458
 InitApplication() function, 36, 53–54
 InitHashTable() member function, 125
 Initializing applications, 32
 InitInstance() function, 35–37, 54
 Inplace activation, 548
 Interface maps, 467
 building, 477–478
 COleObjectFactory class, 495–497
 CoMath class, 484–485
 declaring structures and functions, 478–479
 macros, 475–479
 Interface negotiation, 441–443
 Interfaces, 437–440, 585–586
 communicating with COM
 classes, 438
 custom, 454–455
 early binding, 587
 immutability, 438
 implementing, 485–487
 monikers, 555–557
 multiple, 454–464
 OLE connections, 635
 proxy, 445–446
 stub, 445–446
 InterlockedDecrement() API function, 101
 InterlockedIncrement() API function, 101, 103
 InternalAddRef() function, 467, 471, 473–474
 InternalQueryInterface() function, 467, 471, 473–474
 InternalRelease() function, 467, 471, 473–474
 IOleClientSite interface, 548, 552

- I**
IOleContainer interface, 556
IOleInPlaceActiveObject interface, 438, 560–561
IOleInPlaceObject interface, 438, 560
IOleInPlaceSite interface, 548, 553–554
IOleItemContainer interface, 556–557
IOleObject interface, 438, 549, 558–559
IOleObject::DoVerb() function, 568–574
IOleWindow interface, 553
IParseDisplayName interface, 556
IPC (inter-process communication) facilities, 501
IPersist interface, 454, 458 definition, 442 implementing functions, 486–487 member functions, 442–443
IPersistFile interface, 547, 555
IPersistStorage interface, 438, 547, 549, 558
IPersistStream interface, 547
IStorage interface, 543–544, 547
IStream interface, 544–545, 547
IUnknown interface, 439–440, 457, 467–471 AddRef() function, 440–441 implementing functions, 473–474 interface negotiation, 441–443 QueryInterface() function, 442–443 Release() function, 440–441
IXFWIN, 662

Johnson, Ralph, 59

Keys, 122 sequences from users, 19
Kindel, Charlie, 643
Kirtland, Mary, 584

Late binding, 587
Layer of abstraction, 3–4
LBN_CHANGESEL message, 81
LB_ADDSTRING message, 214
LB_GETITEMDATA message, 254
LB_SETITEMDATA message, 254
Licensing, 26
Linked lists, 14

List boxes, 19, 253–255
Lists, 108, 114–121 graphical, 19 in views, 23
Local servers, 445
Locking string buffer, 96
LockServer() function, 447
LPARAM parameter, 67

Macros, 153, 475–479 different versions of, 414–417 exception-handling, 143 interface map, 475–479 message maps, 71–75 type libraries, 615
Main application, 30
Main window of application, 37
MainWndProc() procedure, 35
MAPI (Messaging Application Programming Interface), 11, 28
Mapping window handles to objects, 44–45
Maps, 14–15, 108, 122–129, 173 associations, 122, 126–128 hashing, 125–129 implementing, 123–124 key, 122 size ceiling, 125–126 value, 122
Marshaling, 446
McCrory, Dean, 99–100, 119, 121, 416, 660
MDI (Multiple Document Interface) applications base class, 18 child window, 18, 56 frame windows, 55–56 handling documents, frames, and views, 264–265
Member functions calling for regular window message, 88–89 classes, 5 structs and, 5
Memory accessing block, 139 allocation, 192 CArray classes allocation, 110 checking allocations and deallocations, 178
CList class management, 120–121
CMemFile handling, 138–139 exceptions, 13 files, 136 implementing shared, 129–130 passing objects to user-provided functions, 195 reviewing statistics, 179 serializing to block of, 136–138 testing maximum usage, 178
Memory checkpoints, 14
Memory diagnostics, 14 advanced, 187–192 checking allocations and deallocations, 178
CObject class, 177–179 generating list, 195
Menus, 20, 263–265 _messageEntries array, 72, 74
Message filter hook, 57
Message handling, 16, 29, 32, 37, 65–92 activating windows, 90–91 C++, 69–70 C/SDK, 38, 67–69 CCmdTarget class, 65 general control notification, 89–90 hooking into message loop, 91–92 message maps, 65–66, 70–91 MFC (Microsoft Foundation Classes), 38 regular window messages, 87–89 window messages, 66–67 WM_COMMAND message, 81–87
Message loop, 16–17, 32, 54 C/SDK, 37 hooking into, 91–92 MFC (Microsoft Foundation Classes), 37 processing window messages before getting to windows or targets, 91 removing tool tips from screen, 91
Message maps, 16, 38, 65–66 actual entries into table, 70–71 AFX_MSGMAP structure, 71

- AFX_MSGMAP_ENTRY**
 structure, 70–71
- beginning**, 73
- CCmdTarget class**, 70
- CTestView class**, 72
- CView class entry**, 302
- data structures**, 70–71
- ending**, 73
- entry**, 280
- finding out if window is MFC window**, 78
- function signatures**, 83–85
- hash table**, 81
- inheritance**, 74
- macros**, 71–75
- message types**, 75
- representing**, 71
- Message pump**, 58, 67–68
- Message reflection**, 90
- MessageBox() function**, 18
- MessageMapFunctions union**, 88–89
- Messages**
- commands as, 75, 81–85
 - controls, 81
 - filtering out, 80–81
 - path for commands, 85–87
 - regular window, 75, 87–89
 - request for on-line help, 81
 - retaining record of last, 78
 - values stored in, 9
- Metafiles**, 20, 61
- METHOD_PROLOGUE() macro**, 487
- MFC (Microsoft Foundation Classes)**
- application framework classes, 12, 21–22
 - automation and, 602–617
 - basic application components, 39–54
 - canned components, 10–11
 - class factory support, 487–498
 - COM (Component Object Model) classes, 464–467
 - CoMath class creation, 467
 - compatibility and portability, 11
 - design goals, 8–10
 - diagnostics in versions, 180
 - document/view architecture, 21–22
 - encapsulating packing and unpacking parameters, 9
- events**, 641–642
- exception macros**, 144
- extendability**, 10
- general-purpose classes**, 12–15
- high-level abstractions**, 12, 22–24
- high-level architecture support**, 10–11
- initializing particular instance of application**, 37
- interface threads**, 58
- macros**, 475–479
- main window of application**, 37
- MAPI (Messaging Application Programming Interface)**, 11
- message handling**, 37–38
- multiple interfaces**, 474
- operating system extensions**, 24–28
- real-world applications**, 8–9
- showing window**, 43
- small and fast**, 10
- type information**, 614–617
- version 1.0, 10**
- version 2.0, 10–11**
- version 2.5, 11**
- version 3.0, 11**
- version 3.1, 12**
- version 4.0, 12**
- vs C/SDK**, 30–38
- Windows API classes**, 12, 16–20
- WinMain() function**, 37
- WinSock support**, 11
- Microsoft C/C++ version 7.0 and MFC (Microsoft Foundation Classes) version 1.0, 10**
- Microsoft Software Development Kit**
- See* **SDK**
- Mini windows**, 392–393
- MINIMAL C (C/SDK) listing**, 32–34
- MINIMAL CPP (C++/MFC) listing**, 34–35
- MinimalCWClass class**, 36
- Mirror file**, 298
- Modal dialogs**, 201–203
- creation, 207–210
 - manipulation, 203
- Modal property sheet**, 241
- Modeless dialogs**, 201–204
- creation, 210–211
- destroying**, 203
- displaying**, 203
- Modularity**, 1, 3–4, 6
- Module state**, 400–403
- Modules**
- AFX_MODULE_STATE structure**, 52
 - core information about, 50
 - grouping classes into, 3
 - retrieving file name, 53
- MoveWindow() function**, 18, 43
- MRU (most recently used) file list**, 394–396
- MSG structure**, 91–92
- Multiple inheritance**, 455–458
- Multiple interfaces and nested classes**, 474
- Native window handles (HWNDs)**, 44
- Nested classes**, 458–466, 474–476
- new operator**, 203
- NewNode() function**, 118
- NMHDR structure**, 89
- Nontemplate collection classes**, 109
- Object dumping**, 184
- Object-oriented programming (OOP)**
- reasons to use, 6–7
 - reusable application frameworks, 7
 - terminology, 2–3
- Objects**, 3–4
- C++ and, 4–6
 - class name at run time, 153
 - derived, 14
 - drawing, 20
 - dynamic creation, 13
 - encapsulated, 4
 - essential distinguishing characteristics, 2
 - inheritance, 4
 - layer of abstraction, 3–4
 - lifetime management, 440–443
 - linked list of CString, 14
 - mapping string to derived COObject object, 15
 - mapping window handles to, 44–45
 - modularity, 4
 - outputting information about, 13
 - parent at run time, 153

- polymorphism, 4
 real-world modeling, 6
 representing windows, 42, 44
 storing and recovering states later, 162–173
 updating user-interface objects, 16
 validity checking, 176
ODBC (Open Database Connectivity), 10–11, 27
ODL (Object Description Language), 597–599
OLE (Object Linking and Embedding), 434, 503–504
 automation, 25
 base classes, 26
 class factories, 25
 clients and exceptions, 13
 compound documents, 24–25
 controls, 26
 memory allocation, 443–444
 outgoing interfaces to other objects, 26
 remote procedure calls, 445–446
 support, 24–26
UDT (Uniform Data Transfer), 25–26, 501, 504–538
 version 2.0, 10–11
OLE Clipboard, 509–514
OLE common dialogs, 236–239
OLE connections, 634–639
OLE controls, 619–622
 classes, 624–628
 constructing, 633
 containers, 628–629
 creation, 630–632
 defining property, 648
 dialog boxes, 622–631
 events, 639–641
 final setup, 633–634
OLE connections, 634–639
 property pages, 646–653
 stand-alone, 631–632
 views, 623
 writing, 621–622
OLE document server, 465
OLE documents, 539–542
 adding container items, 562
 connecting server and container windows, 570–573
 containers, 548–549
 content objects, 547
 deactivating items, 575–577
 embedding, 540–542, 561–574
 in-place activation, 548, 568–574
 in-place frame creation, 569–573
 inserting new item, 563–567
 linking, 540–542
 loading, 578–580
 menus and toolbar setup, 573–574
 MFC container support, 552–557
 MFC support, 550–551
 new files and container, 561–562
 protocol, 550
 saving container's document, 577–578
 servers, 549, 557–561
 Structured Storage, 542–547
OLE drag-and-drop, 526–527
 classes, 531–534
 data consumer, 536–538
 data producer, 534–536
 drop target implementation, 529–531
 handling notification, 537–538
 implementing, 528
 originating, 528–531
 registering view as drop target, 529
 registering with OLE, 536–537
 revoking registration, 537
OLE server exceptions, 13
OLEDLG DLL file, 236
OLEUIPASTESPECIAL
 structure, 238
OleUIPasteSpecial() OLE API
 function, 238
On-line help, 22
OnBeginPrinting() function, 300, 304
OnChildNotify() function, 90
OnDragEnter() function, 529–531
OnDragLeave() function, 529–531
OnDragOver() function, 529–531
OnDraw() member function, 21, 264, 299
OnDrop() function, 529–531
OnEndPrinting() function, 300, 306
OnFileNew() function, 267
OnFileOpen() function, 267
OnFilePrint() function, 302–304, 306
OnIdle() function, 42
OnLButtonDown procedure, 38
OnNotify() function, 90
OnOpenDocument() function, 285
OnPaint() function, 44, 63
OnPrepareDC() function, 300, 306
OnPreparePrinting() function, 300, 303
OnPrint() function, 300
OnRenderData() function, 515–516
OnRenderFileData() function, 515–516
OnRenderGlobalData() function, 515–516
OnSaveDocument() function, 287
OnWndMsg() function, 87–90
ON_COMMAND macro, 74–75
ON_COMMAND_RANGE macro, 75
ON_CONTROL_RANGE macro, 75
ON_MESSAGE macro, 75
ON_REGISTERED_MESSAGE
 macro, 75
ON_UPDATE_COMMAND-
 UI RANGE macro, 75
ON_UPDATE_COMMAND_UI
 macro, 75
ON_WM_CTLCOLOR macro, 90
ON_WM_LBUTTONDOWNCLK
 macro, 74
Open() function, 298
OpenClipboard() function, 502–503
OpenDocumentFile() function, 266
OPENFILE structure, 227
OPENFILENAME structure, 227
Operating system extensions
 DAO (Data Access Objects)
 support, 27–28
 MAPI support, 28
ODBC (Open Database
 Connectivity) support, 27
OLE (Object Linking and
 Embedding) support, 24–26
 Pen Windows support, 28
 WinSock support, 28
Operator []], 114, 127–128
Out-of-proc servers, 444–446
 class factories, 449–450
 unloading, 451–452
OutputDebugString() function, 183
Owner-draw controls, 19

- Pages viewport, 22
 Painting, 61
 Palettes, 62
 Panes and views, 363
 Pen Windows, 28
 Pens, 62
Persistence, 13, 162–173
 control bars, 382–390
Persistent objects, 547
Person class, 5–6
Person struct, 4–5
 Picture properties, 26
POINT structure, 15
 Pointers, 14–15
Polymorphic, 69–70
 Polymorphism, 1, 3–4, 6, 136
POSITION pointer, 115, 121–122
PreTranslateMessage() function, 37
Print dialog box, 300, 304
 Print preview, 299, 307–316
PRINTDIALOG structure, 303
PRINTDLG structure, 302
 Printers, 18
Printing, 18, 299–306
 canceling, 304
 state of classes, 175
Process state, 400–401
Programs
 memory checkpoints, 14
 parsing starting command line, 17
 structured design, 1
 thread of execution, 16–17
 tracking memory leaks, 14
 unsupported feature exceptions, 13
Progress bar, 19
Properties verb, 649–651
Property pages
 assessing properties, 652–653
 dialog template, 648
 linking dialog box control to,
 648–649
 OLE controls, 646–653
 programming, 648
Property Sheets, 19, 239–250
 common control, 243
 controls, 20
 flags, 243
 messages to retrieve information, 244
 notifications, 243–244
 versions 3.0 and 4.0, 242–243
PropertySheet() API function, 243
PropertySheet::Create() function, 240
PROPSHEETHEADER structure,
 243, 245, 248
PROPSHEETPAGE structure, 243,
 248–249
Protocols, 2
Proxy, 445–446
PSM_ADDPAGE message, 247
Pushbutton, 19

QueryInterface() function, 441–443,
 457

Real-world
 applications, 8–9
 modeling, 6
Realloc() function, 139
RECT structure, 15
Reference counting, 96, 98–105
Regions, 63
RegisterClass() API function, 31, 68
RegisterDragDrop() function, 526
 Registering window classes, 54–57
Registry and adding keys, 491–493
Regular child windows, 55
Regular frame windows, 25
Regular window messages, 75, 87–89
Release() function, 457
ReleaseFile() function, 295
 Remote procedure calls, 445–446
 Remote servers, 445
RemoveView() function, 290
Resources
 DLL (Dynamic Link Library),
 408–414
 failure to read exception, 14
 naming symbols, 658
Reusable application frameworks, 7
Rich edit control, 20
Rich Text Edit control, 23
Root class, 13
RTCI (run-time class information),
 153–159
RTF (Rich Text Formatting)
 formatting, 20
RTTI (run-time type information)
 language, 154
Run() member function, 37, 58
Run-time
 discoverability, 441–443
 error checking, 175–176
 type identification, 13
RUNTIME_CLASS macro, 158

Safe user-interface threads, 58
Scribble application, 263
Scroll bar, 19
Scroll views, 22–23
SDI (Single Document Interface)
 applications
 base class, 18
 handling documents, frames, and
 views, 264, 265
 registering frame windows, 56
SDK (Software Development Kit), 10
Searching and replacing text, 18
Searching strings, 95
Sentinels, 192, 194
Serialization, 13, 162–173
 abstract base classes, 173
 arrays, 173
 block of memory, 136–138
CObject class, 169–172
 correct macros for, 170
 declarations, 163–169
 large archives of data, 173
 maps, 173
 performance, 173
 read operation steps, 172
 store operation, 171
Servers
COM (Component Object Model)
 classes, 444–445
 compound documents, 25
 embedded or linked OLE item, 24
 in-proc, 444–445
 local, 445
 OLE document, 549, 557–561
 out-of-proc, 444–446
 reference count, 493–494
 registration, 489–491
 remote, 445
SetClipboardData() function, 502

- S**etCurrentHandles() function, 53
SetData() member function, 5
SetErrorMode() function, 52
SetHandle() member function, 142
SetLoadParams() API function, 173
SetMaxPage() function, 302
SetMinPage() function, 302
SetSize() function, 110
SetStoreParams() API function, 173
Setting up applications, 32
SetWindowLong() function, 77
SetWindowsHookEx() function, 57
Shared Memory, 129–130
SHAREVISTRING macro, 233
ShowData() member function, 5–6
ShowWindow() function, 18, 42–43
Simple value type utility classes, 93–107
Singly rooted hierarchy, 151
SIZE structure, 15
Sliders to choose between range of values, 20
Smalltalk MVC (model-view-controller) architecture, 261
Software patterns, 59–60
Source code files, 6
Spell Check dialog box, 202
Spinners, 20
Splitter windows, 23, 333–365
 dynamic, 23, 333–336
 hit testing, 358–360, 363
 panes, 334–357
 parts of, 334
 scrollbars, 335
 split box, 354–355
 splitter border, 334
 static, 23, 333, 336
 synchronization, 334–335
 tracking, 360–363
SPLS_DYNAMIC_SPLIT style, 345
Stand-alone OLE controls, 631–632
Standard controls, 19
Standard threads, 58
Standard Windows programs, 17
States
 information, 48–51
 module, 400–403
 process, 400–401
 relatedness, 405–406
 thread, 400, 403–406
Static class members, 155
Static control, 19
Static splitter windows, 23, 333, 336
Status bar, 20, 24
STGMEDIUM structure, 506–507
Stream pointer, 134
String buffer, 96
String-izing (#) macro, 157
StringFromClassID() function, 493
Strings
 copying data before write operation, 104–105
 limiting copying data, 100–105
 list of, 115–118
 mapping, 15
 printf-style variable arguments, 95
 searching, 95
Structs, 2, 4–5
 defining, 4
Structured Storage
 compound files, 546–547
 ILockBytes interface, 545–546
 IStorage interface, 543–544
 IStream interface, 544–545
 persistent objects, 547
 storages, 542
 streams, 542
Sub, 445–446
Subtract() function, 455, 458
Swap tuning, 660–661
Synchronization object class, 17
System-defined window handle, 77
 _t string routines, 105–106
Stabbed dialogs, 19, 239–250
StagRECT structure, 106, 107
Stask Allocator, 585
Stemplate collection classes, 109
Stext files, 133
Sthin wrapper, 28
SThread state, 400, 403–406
SThread-safe classes, 17
Sthreads, 417–429
 encapsulation, 58
 message queue, 429
MFC interface, 58
safe, 58
standard, 58
user-interface, 16, 424–429
Win32, 58
 worker, 16, 58, 417–424
3D Controls, 77
THROW macro, 144
THROW_LAST macro, 144
Time, 15
Tool tips, 20
Toolbars, 20, 23–24, 56
TRACE macro, 174–175, 180–181
TRACE3 macro, 147
Transferring data
 advisory connection between data object and advisory sink, 509
 caching data, 519–520
 data consumers, 5–6, 511
 data producer, 506, 511
 delayed rendering, 515–516
 describing data, 504–506
 disconnecting advisory sink, 509
 enumerating advisory connections, 509
 format and medium, 508
FORMATEC structure, 504–506
 formats data can be stored or retrieved, 509
 holding data, 506–507
 old way, 502–504
OLE Clipboard, 509–514
 poking data into data object, 508
 retrieving data, 508
STGMEDIUM structure, 506–507
TranslateMessage() function, 37, 58
TRY macro, 144
Two debug new operators, 189–190
Type information
 exploiting, 599–600
 loading type library, 599
MFC (Microsoft Foundation Classes), 614–617
Type libraries, 615–617
Typecasting, 9

- UDT (Uniform Data Transfer), 25–26, 438, 501, 504–538
IDataObject interface, 507–509
 OLE Clipboard, 509–510
 OLE drag-and-drop, 526–538
UNC (Uniform Naming Convention), 133
Uncommon memory pattern, 192
Undocumented CPlex structure, 116–117
Undocumented state information, 48–51
Unsigned integers, 14
Unsupported program feature exceptions, 13
UpdateAllViews() function, 288
User-interface objects, 16
User-interface threads, 16, 424–429
Users
 choices in dialogs, 271–273
 key sequences from, 19
 notifying that exception has occurred, 146–148
USR DLLs, 407–408
Utility classes
 collection classes, 107–129
 exception-handling, 143–148
 files, 129–143
 simple value type, 93–107
Value, 122
Variables, 657
VARIANT data type, 590, 593
VBXs (Visual Basic controls), 26
 shortcomings, 620
Views, 261–264
 adding event sink, 642–646
 applying different user interface to, 264–265
 communicating with, 287–288
 controls from dialog template, 324–327
 creation, 279–281, 325–327
 drawing, 288–289
 enhanced, 22
 Explorer-like tree, 23
 initialization, 325–327
 lists in, 23
 OLE controls, 623
 panes, 363
 scrolling, 317–319
 updating in frame, 281
 Windows control as, 327–330
Virtual functions, 3, 69–70, 198
Visual Basic controls, 19
Visual C++
 browser, 661
 Find in Files, 662
 new OLE classes, 10
 syntax-coloring tip, 662
 version 1.0 for NT, 11
 version 1.0 for Windows, 11
 version 1.5, 11
 version 4.0, 12
Vlissides, John, 59
Wait cursor, 17
Win32 API, 43
Win32 threads, 58
WINDEF H header file, 106
Window classes, 54–57
Window handles, 44–46
Window messages, 44, 66–67
Window objects, 44–46
WindowProc() member function, 38, 87
Windows, 30
 accepting dragging and dropping transfers, 26
 activating, 90–91
 application requirements, 32
 C++ class representing, 77
 class derived from, 18
 client area, 20, 61
 controls, 19
 defining basic aspects and behavior, 54–57
 encapsulating device context, 20
 entire screen area, 61
 finding out if MFC window, 78
 first to receive messages, 18
 for SDI or MDI child window, 55
 frame, 18
 message handling procedure, 76–77
 mini, 392–393
 object representing on-screen, 42, 44
 panes, 23
 registering, 56
 regular messages, 75
 showing, 43
 system-defined window handle, 77
 viewing data, 21–22
Windows (Microsoft Windows 3.1 and Windows95)
 API classes, 12, 16–20
 application developers, 7
 basic application support, 29–30
 Clipboard, 501–503
 common dialogs, 227
 control classes, 250–257
 DDE (Dynamic Data Exchange), 501, 503
 event-driven, 31
 hooks, 57–58
 IPC (inter-process communication) facilities, 501
 metafiles, 61
 objects, 62
 old way of transferring data, 502–504
 programming overhead, 31
 programs, 17
 resource failure to read exception, 14
 simplifying API (Application Programming Interface), 9
 steps to qualify as Windows program, 31–32
 visual interface objects, 18
Windows Clipboard, 501–503
Windows NT, 10
WinMain() function, 16–17, 29, 31, 35–36, 46–48, 51–52, 54, 58
C/SDK, 36
MFC (Microsoft Foundation Classes), 37
WinSock, 11, 28
WM_ACTIVATE message, 90, 185
WM_ACTIVATETOLEVEL message, 90
WM_COMMAND message, 38, 67, 69, 75, 80–89, 263, 428, 477
WM_COMMANDHELP message, 81

- WM_CREATE message, 280
WM_CTLCOLOR message, 77, 90
WM_DESTROY message, 32, 38
WM_DRAWITEM message, 254
WM_INITDIALOG message, 78, 206,
 212, 214
WM_INITIALUPDATE message, 281
WM_LBUTTONDOWN message, 32,
 38, 528
- WM_MOUSEMOVE message, 66, 75
WM_MOVE message, 89
WM_NCPAINT message, 393
WM_NOTIFY message, 80, 89–90,
 243, 249
WM_PAINT message, 20–21, 44, 263,
 288, 428
WM_QUERYAFXWNDPROC
 message, 77–78
- WM_QUIT message, 36–37
WM_SETCURSOR message, 80–91
WM_SIZE message, 56, 87–89, 263
WNDCLASS structure, 56, 68
WndProc() function, 77
Word values, 14
Words, 15
Worker threads, 16, 58, 417–424
WPARAM parameter, 67

MFC Internals

Inside the Microsoft® Foundation Class Architecture

"This book is definitely not a rehash of existing documents. It is not a 'how-to' book—it is a 'how does it work' book." —Dean McCrory, MFC Development Lead

Finally, a book on MFC that fills the gaps between "Using the Wizards" Visual C++ books, product documentation, and MFC source code. *MFC Internals* is a guide to what goes on inside the Microsoft® Foundation Classes: It gives you unique and in-depth information on undocumented MFC classes, utility functions and data members, useful coding techniques, and critical analysis of the way various MFC classes work and fit together.

The first half of the book covers core Windows® graphical user interface classes and their supporting classes; the second half covers subjects like OLE that are extensions to the basic Windows support. You'll become an expert at understanding MFC implementation details by:

- exploring under the hood of MFC's document/view architecture to learn about view synchronization, printing, and even print preview
- diving deep into undocumented aspects of MFC serialization and undocumented classes like CPreview, CPreviewDC, CMirrorFile, CDockBar, and so on
- finally learning how MFC and OLE work together, and how OLE controls are implemented
- building the skills that help you investigate and understand MFC source code on your own.

MFC Internals focuses on MFC 4.0 for Windows 95 and Windows NT. Most key "internal" concepts also apply to previous versions, but where they don't the authors warn you with a version note. The book's disk contains example code and the MFC FAQ file. Be sure to check out Appendix A, a handy MFC source code field guide.

MFC Internals is an essential guide to tapping MFC's rich and robust application framework and applying advanced MFC knowledge in world-class Windows applications.

George Shepherd is a senior computer scientist with DevelopMentor, where he develops and delivers courseware for developers using MFC and OLE.

Scot Wingo is a cofounder of Stingray Software, an MFC extension company. He is also the maintainer of the MFC FAQ <http://www.unx.com/~scot/mfc_faq.html>

Cover design by Jean Seal in the style of Mark Rothko

Addison-Wesley Publishing Company, Inc.

Pearson Education

<http://www.awl.com/cseng/>

