INSIDE OLE 2

ABB Inc.

EXHIBIT 1011

INSIDE
OLE 2

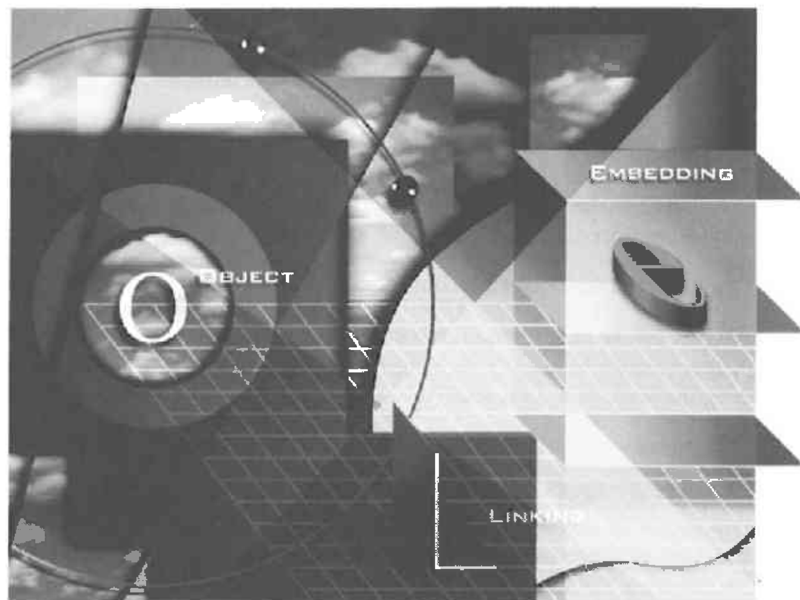
The Fast Track

to Building

Powerful

Object-Oriented

Applications



KRAIG BROCKSCHMIDT

Microsoft
PRESS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 1994 by Kraig Brockschmidt

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Brockschmidt, Kraig, 1968–
Inside OLE 2 / Kraig Brockschmidt.
p. cm.
Includes index.
ISBN 1-55615-618-9
1. Object-oriented programming (Computer science) 2. Microsoft Windows (Computer file) I. Title.
QA76.64.B76 1993
005.4'3--dc20 93-34953
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 AGAG 9 8 7 6 5 4

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Microsoft and MS-DOS are registered trademarks and Visual C++, Windows, and Windows NT are trademarks of Microsoft Corporation. Apple is a registered trademark and Macintosh is a registered trademark of Apple Computer, Inc. Borland is a registered trademark of Borland International. Smalltalk is a registered trademark of Xerox Corporation.

Acquisitions Editor: Dean Holmes
Project Editor: Ron Lamb
Technical Editor: Seth McEvoy

CONTENTS

Preface ... xv
Using the Companion Disks ... xxii

SECTION I
WINDOWS OBJECTS

CHAPTER 1
AN OVERVIEW OF OLE 2 ... 3
Windows Objects: The Component Object Model ... 5
Objects and Interfaces ... 8
Structured Storage and Compound Files ... 12
Uniform Data Transfer and Notification ... 15
- Notification ... 17
- Data Objects and the Clipboard ... 17
- Data Objects and Drag-and-Drop ... 17
- Data Objects and Compound Documents ... 18
- Data Objects and DDE ... 19

Compound Documents: Object Embedding ... 19
Compound Documents: Object Linking and Monikers ... 21
Compound Documents: In-Place Activation (Visual Editing) ... 23
Automation ... 24

CHAPTER 2
CONVENTIONS, C++, AND SAMPLE CODE ... 27
To C or Not to C (with Apologies to Shakespeare) ... 27
- User-Defined Types: C++ Classes ... 29
- Access Rights ... 33
- Single Inheritance ... 34
- Virtual Functions and Abstract Base Classes ... 37
- Multiple Inheritance ... 39

Sample Code ... 41
- Include Files: The INC Directory ... 42
- Libraries: The LIB Directory ... 43
- The BUILD Directory ... 43
- Three Amigos: BttnCur, GizmoBar, and StatStrip ... 44
- Class Libraries: The CLASSLIB Directory ... 45

Interface Templates: The INTERFAC Directory 48
Chapter Sources: The *CHAPxx* Directories 48
Cosmo: A Graphical Editor (with Apologies to No One in Particular) .. 49
Patron: A Page Container (with Apologies to Merriam-Webster) 51
Building and Testing Environment 54

CHAPTER 3

OBJECTS AND INTERFACES **57**
The Ultimate Question to Life, the Universe, and Objects (with Apologies to Douglas Adams) 58
Windows Objects vs. C++ Objects 64
Let's Go Traveling 64
Other Differences Between Windows Objects and C++ Objects 67
A Simple Object in C and C++: *RECTEnumerator* 70
RECTEnumerator and the *IEnumRECT* Interface 72
Creating the *RECTEnumerator* Object 94
Using an *IEnumRECT* Pointer 95
Reference Counting 96
My Kingdom for Some Optimizations! 98
Call-Use-Release 100
IUnknown, the Root of All Evil 101
QueryInterface vs. Inheritance 103
QueryInterface Properties and Interface Lifetimes 104
Some Data Types and Calling Conventions 106
STDMETHOD and Associates 106
HRESULT and SCODE 107
Globally Unique Identifiers: GUIDs, IIDs, CLSIDs 109
OLE 2 Interfaces and API Functions 110
Custom Interfaces 112
Interfaces vs. API Functions 112
What Is a Windows Object? (Reprise) 113
Summary 114

CHAPTER 4

COMPONENT OBJECTS (THE COMPONENT OBJECT MODEL) **117**
Where the Wild Things Are (with Apologies to Maurice Sendak) 119
The New Application for Windows Objects 121
Enlarge the Message Queue 122
Verify the Library Build Version 123
Call *CoInitialize* or *OleInitialize* 124
Call *CoUninitialize* or *OleUninitialize* 125

Memory Management and Allocator Objects 126
Component Objects from Class Identifiers: A Component User 138
#include guid.h and Precompiled Headers 148
Instantiate a Component Object 149
Manage the Object and Call *CoFreeUnusedLibraries* 152
Implementing a Component Object and a Server 153
Register CLSIDs 174
Implement the Class Factory 176
Expose the Class Factory 178
Provide an Unloading Mechanism 182
Cosmo's Polyline as a DLL Object 187
Object Reusability 191
Case 1: Object Containment 194
Case 2: Object Aggregation 195
Summary 201

SECTION II

OBJECT-ORIENTED SYSTEM FEATURES: FILES AND DATA TRANSFER

CHAPTER 5

STRUCTURED STORAGE AND COMPOUND FILES **205**
Motivation 208
Patron Files with the Jitters 209
The Decaffeinated Alternative 211
Energy Boosts Without the Jitters: Compound Files 213
Features of Compound Files 215
Stream, Storage, and *LockBytes* Objects 215
Element Naming 217
Access Modes 217
Transacted Storages 219
Incremental Access 220
Shareable Elements 222
Compound File Objects and Interfaces 223
Storage Objects and the *IStorage* Interface 223
Stream Objects and the *IStream* Interface 227
LockBytes Objects and the *ILockBytes* Interface 229
Compound Files in Practice 233
Simple Storage: Cosmo 234
Pulling Rabbits from a Hat with STGM_CONVERT 240
Streams vs. Files 241

Complex Compound Files: Patron 244
The Root Storage and Temporary Files 263
Managing Substorages 264
Multilevel Commits 265
File Save As Operations 268
Low-Memory Save As Operations 269
Streams as Memory Structures 271
Other OLE 2 Technologies and Structured Storage 272
The *IPersistStorage*, *IPersistStream*, and *IPersistFile* Interfaces 273
A Heavy Dose of Protocol with *IPersistStorage* 275
Of Component Users and *IPersistStorage*: Component Cosmo 278
Of Component Objects and *IPersistStorage*: Polyline 281
Compound File Defragmentation 288
Summary 297

CHAPTER 6

UNIFORM DATA TRANSFER USING DATA OBJECTS **299**
What Is a Data Object? 301
New and Improved Ultra-Structures! 302
Data Objects and the *IDataObject* Interface 308
FORMATETC Enumerators and Format Ordering 312
Component Data Objects 318
Some *CDataObject* Features 329
Implementing *IDataObject* 330
A (Component) Data Object User 331
Advising and Notification with Data Objects 345
Establishing an Advisory Connection 351
Sending Notifications as a Data Object 355
Special Considerations for Remoted Notifications 359
Inside the Advise Sink 360
IDataObject as a Standard for Object Data Transfer 363
View Objects and the *IViewObject* Interface 365
IViewObject::Draw 367
Rendering for a Specific Device 369
Drawing into a Metafile 370
Aborting Long Repaints 370
Other *IViewObject* Member Functions 371
IViewObject and Notification 372
Freeloading from OLE2.DLL 373
IDataObject and DDE 383
Summary 385

CHAPTER 7

CLIPBOARD TRANSFERS USING DATA OBJECTS **387**
The OLE 2 Clipboard Protocol 389
But All I Want to Do Is Copy Some Simple Data! 391
A Data Transfer Component Object 393
If You Already Have a Data Object...Component Cosmo 405
If You Already Have Extensive Clipboard-Handling Code 407
Simple Data Source and Consumer: Cosmo 408
Startup/Shutdown 408
Copy/Cut 409
Enabling Edit/Paste 411
Paste 412
Paste Special and a Functional Patron 414
The Paste Special Dialog Box and the OLE2UI Library 416
Tenant Creation, Paste 420
Saving and Loading Tenants 424
Copy and Cut 425
Summary 426

CHAPTER 8

DRAG-AND-DROP OPERATIONS USING DATA OBJECTS **429**
Sources and Targets: The Drag-and-Drop Transfer Model 430
A Step-by-Step Drag-and-Drop Implementation: Cosmo 439
Design and Implement Drop Target User Feedback 454
Implement a Drop Target Object and the *IDropTarget* Interface 456
IDropTarget::DragEnter 456
IDropTarget::DragOver 459
IDropTarget::DragLeave 460
IDropTarget::Drop 461
Register and Revoke the Drop Target Object 462
Design and Implement Drop Source User Feedback 464
Determine the Pick Event 464
Implement a Drop Source Object and the *IDropSource* Interface 465
Call *DoDragDrop* 466
Intermission 468
Advanced Drag-and-Drop: Feedback and Scrolling in Patron 469
Tenant Pick Regions and Drop Sourcing 470
More Advanced Drop Target Hit-Testing 472
A Feedback Rectangle 474
Scrolling the Page 477
Summary 482

SECTION III

COMPOUND DOCUMENTS: OLE

CHAPTER 9

COMPOUND DOCUMENTS AND EMBEDDED CONTAINERS 487

Compound Document Mechanisms 489
The Passive State 490
The Loaded State 491
Case 1: *InProcServer* 492
Case 2: *InProcHandler* 493
Case 3: The Default Handler 494
Loading the Object: All Cases 496
Drawing the Object 497
The Running State 500
Mommy, Daddy, Where Do New Objects Come From? 502
The Structure of a Container Application 504
Embedding Containers Step by Step 506
Call Initialization Functions at Startup and Shutdown 507
Define Sites and Manage Site Storage 507
Implement Site Interfaces and Add Site Variables 511
Implement *IAdviseSink* 516
Implement *IOleClientSite* 519
Implement Site Shading 525
Invoke the Insert Object Dialog Box 527
Call *OleUIInsertObject* 530
Call *OleCreate* or *OleCreateFromFile* 532
Initialize the Object 534
Draw and Print Objects 538
Activate Objects and Add the Object Verb Menu 539
Mouse Double-Clicks 542
Object Verb Menu 543
The Right Mouse Button Pop-Up Menu 547
Create Objects from the Clipboard and Drag-and-Drop Transfers 549
Copy and Source Embedded Objects 552
Close and Delete Objects 555
Save and Load the Document with Embedded Objects 556
Handle Iconic Presentations (Cache Control) 560
Summary 562

CHAPTER 10

COMPOUND DOCUMENTS AND EMBEDDED OBJECT SERVERS (EXEs) 565

The Structure of a Server Application 566
Linking Support and Mini-Servers vs. Full-Servers 568
Version Numbers 569
Installation 569
Embedding Servers Step by Step 570
Call Initialization Functions at Startup and Shutdown 571
Create Registration Database Entries 572
Implement and Register a Class Factory 577
Implement an Initial Object with *IUnknown* 586
Implement the *IPersistStorage* Interface 594
Implement the *IDataObject* Interface 601
Implement the *IOleObject* Interface 610
Modify the Server's User Interface 629
Send Notifications 633
(Full-Servers) Add OLE 2 Clipboard Formats 637
(Optional) MDI Servers, User Interface, and Shutdown 638
Summary 639

CHAPTER 11

IN-PROCESS OBJECT HANDLERS AND SERVERS 641

The Structure of In-Process Modules 641
Why Use a Handler? 644
Why Use an In-Process Server? 646
Why Not Use an In-Process Server? 647
Delegating to the Default Handler 648
IOleObject 649
IDataObject 651
IPersistStorage (on the Cache) 652
IViewObject 652
Implementing an Object Handler 653
Obtain a Default Handler *IUnknown* 656
Expose Default Handler Interfaces in *QueryInterface* 659
Implement *IPersistStorage* 662
Implement *IOleObject::GetExtent* 666
Implement *IViewObject* 668
Synchronized Swimming with Your Local Server 674
Year-End Bonuses 678
Notes on Implementing an In-Process Server 679
Summary 692

CHAPTER 12

MONIKERS AND LINKING CONTAINERS **695**
Will Someone Please Explain Just What a Moniker Is? 698
Moniker Classes 699
Where Do I Get Monikers? 703
Step-by-Step Linking Container 704
Enable Links from Insert Object 705
Enable Linking from Clipboard and Drag-and-Drop Operations 709
Paste Link and Paste Special Commands 711
Drag-and-Drop Linking Feedback 713
Test Your Linking 715
Implement the Show Objects Command 717
Manage a File Moniker, Call *IOleObject::SetMoniker,* and Implement *IOleClientSite::GetMoniker* 722
The Links Dialog Box and the *IOleUILinkContainer* Interface 727
Invoke the Links Dialog Box 744
Update Links on Loading a Document 747
Summary 751

CHAPTER 13

MONIKER BINDING AND LINK SOURCES **753**
Moniker Binding Mechanisms 755
A Simple Linked Object: Single File Moniker 756
A Linked Object with a Composite *File!Item* Moniker 758
Binding a Composite *File!Item!Item!Item!Item...* Moniker 762
Bind Contents 764
The Running Object Table 765
A Simple Link Source: Cosmo 766
Create, Register, and Revoke a File Moniker 767
Provide Link Source Formats in Data Transfer 770
Implement the *IPersistFile* Interface 774
Implement *IOleObject::SetMoniker* and *IOleObject::GetMoniker* 777
Complex Linking and Linking to Embeddings 779
Why Linking to Embedding? 780
Create and Manage the Composite Moniker 782
Source the Composite Moniker 787
Implement a Class Factory for Document Objects with *IPersistFile* 789
Implement *IOleItemContainer* for Each Item Moniker 792
Summary 806

CHAPTER 14

CONVERSION, EMULATION, AND COMPATIBILITY WITH OLE 1 809

The Convert Dialog Box for Containers 810
Support a Convert Menu Item and Invoke the Convert Dialog Box . 813
Handle the Convert To Case 816
Handle the Activate As Case 818
Handle Display As Icon Changes 820
Conversion Between Servers 822
Registration Database Entries for Conversion 823
IPersistStorage Modifications 826
OLE 1 Embedded Object Conversion and Emulation 828
Notes on OLE 1 Compatibility for Containers 833
OLE 1 Server Quirks 833
File Conversion 834
Summary 839

SECTION IV

COMPOUND DOCUMENTS: IN-PLACE ACTIVATION

CHAPTER 15

VISUAL EDITING: IN-PLACE ACTIVATION AND IN-PLACE CONTAINERS 843

Motivations and the Guts of an In-Place Session 845
Where Does It All Start? 846
An Innocent Little *DoVerb* 848
Activating In-Place 849
Manipulations of an Active Object 857
Pulling the Plug: Deactivation 860
Active vs. UI Active and Inside-Out Objects 861
Yes, This Actually Does Work 861
In-Place Container Step By Step 862
Prepare the Container 863
Implement Skeletal In-Place Container Interfaces 865
Activate and Deactivate the Object 875
Mix-a-Menu: Shaken, Not Stirred 877
Negotiate Tool Space 882

Provide In-Place Accelerators and Focus 892
Round the Corners: Other Miscellany 895
Summary 905

CHAPTER 16

IN-PLACE ACTIVATION FOR COMPOUND DOCUMENT OBJECTS **907**

In-Place Objects Step by Step 908
Drivers, Prepare Your Objects 909
Implement Skeletal In-Place Object Interfaces and Object Helper Functions 912
Implement Simple Activation and Deactivation 915
Assemble and Disassemble the Menu 927
Create and Destroy In-Place Tools 934
Manage and Process Accelerators 938
Rounding Third ... and Heading for Home 940
Where In-Place Activation Can Take Us 950
Summary 955

Index 957

PREFACE

Give me a fish and you feed me for a day.
Teach me to fish and you feed me for a lifetime.
A proverb

This is a book about fish. But because without knowing how to catch them, you'd eventually starve, it's also about fishing. The fish are all those pieces of information that you need as a developer in order to exploit OLE 2 features in your application. Teaching you to fish involves describing why the specific pieces you are using were designed and what path they lay toward the future. Of course, you always need a reason to keep fishing even if you're currently well fed, so at the beginning of each chapter I will attempt to motivate you enough to read it.

It has been said that authors write books not so that they will be understood, but so that they themselves understand. Certainly this work has been such an experience for me. When I started working with OLE 2 in the middle of 1992 as part of my job in Microsoft's Developer Relations Group, I saw the technology as merely a way to create applications that support what is called "Compound Documents," as OLE version 1 was. This attitude was well accepted at Microsoft because OLE 2 was a refinement of OLE 1; in fact, the OLE 2 design specifications are organized around a Compound Document core with a number of other technologies hanging off the sides to solve the most critical problems exposed in OLE 1.

For a number of months, I plodded through prerelease information about OLE 2 to create some sample applications to demonstrate compound documents. With the help of various members of the OLE 2 development team, with whom I've worked closely for all this time, I gave a number of classes inside and outside of Microsoft to help others use OLE 2 to create Compound Document applications. In the back of my mind something was telling me that there was much more to OLE 2 than I had originally perceived, but it was very hard to break away from equating OLE 2 and Compound Documents because every available piece of documentation made the two terms synonymous.

In the first few weeks of January, 1993, I started to see that, in the process of solving the most important problems in OLE 1, the OLE 2 architects had actually created a much larger system for object-oriented programming under Windows. I began to see that OLE 2 has technologies that are separate from the true Compound Document technologies. In fact, I started to see exactly how one might use those other technologies without ever coming into contact with Compound Documents. I was not the first person to realize this. In fact, OLE 2 was actually designed this way, but this aspect of the design unfortunately was lost somewhere between the minds of the OLE 2 architects and the actual OLE 2 Design Specification. But I was slowly beginning to rediscover the elegant underlying architecture of the entire group of technologies. My position within Microsoft allowed me to explore OLE 2 in depth and even to browse the OLE 2 sources, letting me truly get "Inside OLE 2."

One Sunday afternoon in mid-January, 1993, while doing something totally unrelated to OLE 2, I achieved what Eric Maffei (editor of *Microsoft Systems Journal*) describes as "OLE Nirvana." All the little subtechnologies in OLE 2 fell into place and I saw clearly, after six months of mental fog, what OLE 2 was all about. I realized that you could exploit very small pieces of OLE 2 in incremental steps and that the best way to communicate the entire vision was to write a book. I quickly fired up my notebook computer and spent the next three hours pounding out the outline. The book you now hold follows that original outline closely.

My goal in writing this book was to provide an organization for OLE 2 in such a way that each chapter depends *solely* on information in previous chapters, with no dependencies on later chapters. Because OLE 2 is not a technology for writing whole applications (because we still use many Windows API functions), I had the luxury of concentrating on OLE 2's features and the way you use those features in your applications. I have presented the material a little at a time, in order to help you solidify your understanding of that building block before moving on. I hope the book takes you on an evolutionary path, on which the work you do early in the book will be reusable in work you do in the later stages.

This same idea is present even within any given chapter, where I have provided finely detailed step-by-step instructions for implementing specific features and where each step depends on the prior steps but not on any later step. This sort of process enables you to add a little code, compile your application, and actually see something working! Personally, I find the incremental feedback of this sort of process extraordinarily motivating. In fact,

it makes programming fun, and that is refreshing in this day and age of "serious" professional programmers. I got into computers because hacking out some BASIC code was exhilarating. I hope I can bring some of that back through this book.

OLE 2 is the first step in the evolution of Windows from the function call–based operating system we have today to an object-oriented operating system in the future. The object model you will learn in this book will be a part of Windows programming for a long time, and I hope it will help you develop a definite edge in your programming career. Because OLE 2 is a first step, it is going to seem utterly alien much of the time. But you need to learn how to fish sometime if you are ever going to feed yourself. While you are learning the skills of a master angler, this book will help you catch enough fish to keep you from starving.

Who Can Use This Book

I mentioned earlier that OLE 2 is not a technology for writing an entire application. To use OLE 2, you must be familiar with how to write an application for Windows. I will not describe how to use any of the existing Windows API functions, nor will I attempt to describe any intricate details about Windows itself. Our focus in this book is strictly on OLE 2.

Therefore, I assume that you are already familiar with programming in the Windows environment and that you have at least a working knowledge of the Windows API. In addition, because we are talking about object-oriented programming here, a knowledge of C++ is helpful, but not required. In fact, C++ knowledge can at times be a hindrance to understanding the object model in OLE 2. Although the samples in this book are written in C++, I've kept them very much like standard C Windows programs. Chapter 2 contains a short discussion of the C++ I use in all the book's samples, from a C programmer's perspective (which was my own perspective when I started writing this book).

This book is not only for programmers, however. Each chapter is structured so that a person who designs application architectures can read the first few sections and understand how the mechanisms in OLE 2 work without having to work through the details of code. The first 5 to 20 pages of each chapter discuss architecture, leaving exact details about writing code to the latter parts of the chapter. So, if you want an in-depth look at how OLE 2 works, read the first section or two of each chapter.

Some Assembly Required

No, we won't use any *assembly* language, but this book does assume that you have an appropriate software development environment installed that includes the following:

- A C++ Compiler such as Microsoft C version 7 or Microsoft Visual C++ version 1. The make files for the samples in this book are specific to Microsoft compilers, so some adjustment will be necessary for other environments.
- The Windows 3.1 Software Development Kit.
- *And most important*: the OLE 2 Software Development Kit. Be sure that the OLE 2 directory is added to your PATH, INCLUDE, and LIB environment variables before attempting to build any samples in this book. You can obtain the OLE 2 SDK from Microsoft for a $50 charge by calling 1-800-227-4679.

Chapter 2 includes more information on creating the right build environment for the book samples specifically.

On Coding Style

As soon as you start reading some of the code in this book, you'll begin to wonder where I developed my coding style.

My coding style, which is unlike any other widely published standard, is what I've personally developed over a number of years to improve (in my mind) code readability as well as to prevent myself from making certain mistakes. For example, when I want to compare a variable to a constant, I always put the constant first—that is, I'll write *if (0L==m_cRef)* instead of *if (m_cRef==0L)*. Like all C and C++ programmers, I've had my share of bugs because I typed = instead of ==. Putting the constant on the left causes a compiler error when you forget the second equal sign. With the variable on the left, you get a legal assignment statement but a very nasty run-time error.

All other stylistic elements have their justifications as well and are used consistently. I've often heard that people prefer consistency over specific styles, so there you have it. Let me also mention that in many of the source listings in this book, I have eliminated lengthy header comments on files and functions that you will see in the actual disk files. This is done simply to save space. Any code within a function, however, will match exactly what you will find on disk.

Acknowledgments

People who have read drafts of this book have repeatedly asked me where I found my inspiration for writing the way I have. Influence has come from many corners, so let me list those sources as well as offer my thanks to the following groups or individuals who have helped create this tome of OLE:

To all the programmers in the trenches who are usually told to do too much with too little information. Without you, I'd have little incentive to write a book like this.

To all those developers inside Microsoft who took the time to formally review this work: Charlie Kindel, Nigel Thompson, Scott Skorupa, Sara Williams, Vinoo Cherian, Craig Wittenberg, Douglas Hodges, Alex Tilles, Mark Bader, Dean McCrory, and especially Nat "Zoinks" Brown—thanks for all your useful and real-world insights.

To the OLE 2 team at Microsoft for all their answers and input, especially to architects Tony Williams and Bob Atkinson.

To all the developers who devoured my draft copies as soon as I could write them and who sent words of encouragement, including Dominic Kyrie, Marc Singer, Marcellus Bucheit, Lars Nyman, Howard Chalkley, and Jim Adam, a total Python Head who reminded me that it was *Patsy* who actually said Camelot was only a model.

To Burt Harris and Thomas Holaday for setting me straight on the finer points of C++ programming.

To Monty Python, Yoda, the *Harvard Lampoon*, and *MAD Magazine*, as well as authors Donald Norman, Robert Fulghum, Tom DeMarco, Timothy Lister, Douglas Adams, Piers Anthony, Marvin Harris, and Jim Stacey for whatever it is that made me include the crazy things I wrote in this book.

To photographer Dewitt Jones, Lynette Sheppard, and the entire group from our week at HollyHock, who showed me how to enjoy and appreciate doing the crazy things I have in this book. May you always fly with frozen eagles.

To Dean Holmes, Ron Lamb, Seth McEvoy, and all the other people at Microsoft Press, not only for doing the work of publishing this book but also for letting me get away with the crazy things I've done here.

To Bob Taniguchi for helping me get into the position at Microsoft to write this book, and to Viktor Grabner for teaching me what the purpose of making my job obsolete really means and how to be a little crazy in the process.

To Microsoft's Developer Relations Group for allowing me to lock myself in my office undisturbed for months on end while I was doing crazy things.

And, of course, to my wife, Kristi, who was there through what has been the busiest year we've yet experienced.

Road Map

Before we get started, let me give you a word of warning. OLE 2 is big. Very big. If you count the number of new functions in OLE 2, you have more than in Windows 3.0 itself. If you count the number of pages in this book, you'll find it longer than the Windows 3.1 edition of Charles Petzold's *Programming Windows* (no offense, Charles), and I don't list most of the sample code. What does this mean for you? If there ever was a time to heed the warning "Don't bite off more than you can chew," now is the time. Allow me to illustrate.

Pat and Casey each decided to build a cabin at the top of a mountain. Each cabin would have the latest siding, a hardwood floor, and a pressure-treated deck with a great view of the valley.

Pat was so excited about actually having this cabin that she quickly threw some wood and tools into a helicopter and went straight to the summit. "Time is money," Pat philosophized, as she started hammering away. Soon she needed another tool and more materials, so she rushed back down the mountain, grabbed what she needed, and hurried back up. This process repeated itself again and again and again. Pat never had the right things on hand to complete the job efficiently, although on every flight back up the mountain she thought she did. But the progress was impressive.

Meanwhile, Casey did not start so quickly. She carefully planned an approach to the construction, organized all the materials she would need, and arranged for them to be delivered just before they were necessary. She intended to have everything on hand to complete each stage in the project.

When Casey eventually arrived at the summit, she had only enough to build the foundation, but it went perfectly. Pat would often peer over and laugh, touting how much more she had accomplished and how much faster. "Time is money!" she would shout. Casey would quietly think, "If time is money, why are you spending as much time going up and down this mountain as you are building?"

It seemed that Pat would complete the project long before Casey. Pat had an insatiable urge to keep building something, and so she was finishing the floor and building parts of the deck with only half the walls and roof complete. Casey had only a foundation, the frame, and the roof completed on her cabin.

Unexpectedly, powerful monsoons fell upon the two builders. Pat watched in horror as the incomplete walls and roof were torn away in the strong winds. She could only stand there helplessly and get drenched, watching her beautiful wood floors split under the intense pounding. Casey, keeping perfectly dry with a solid roof overhead, continued working through the rains, adding wallboard so that her cabin would withstand the wind and completing a magnificent interior, all the time staying warm and dry.

When the monsoons subsided and the sun returned, Pat cleaned up the wreckage and salvaged what she could. Months later, she completed her cabin—it was finished,

but it wasn't what she had imagined. She was just glad that it was finally done.

After the storms, Casey had only to spend a few more weeks on the deck before she finished her cabin. While Pat was painfully recovering what she could, Casey was enjoying a wonderful spring in the mountains.

Catastrophes often occur in software development. A competitor releases a product that has more features than you knew about, sooner than you expected. If your work doesn't stand up to that competition, it has to be scrapped or completely reworked.

The approach taken in this book will help you incorporate technologies of OLE 2 into your application in such a way that you can stop completely at a number of points but will still have a lot to show for your efforts.

Section One, "Windows Objects," discusses the basic architecture of the OLE 2 object model. This in itself does not contain much that will be very visible to your customers, but it is the foundation.

Section Two, "Object-Oriented System Features: Files and Data," describes a new way to read and write disk files that is powerful enough in itself that it might be the only technology you exploit in OLE 2. This new method simplifies features such as incremental saves and transactioning. In addition, Section Two deals with the concepts of Uniform Data Transfer, through which you can gain significant performance benefits, especially within suites of applications. That alone might be enough to satisfy immediate demands of your customers. You might also want to exploit OLE 2's drag-and-drop protocol, on which many features in future versions of Windows will be based.

From there, you can work into Section Three, "Compound Documents: OLE," which explores the concepts and necessary code to support Linked and Embedded objects according to the OLE 2 Design Specification for Compound Documents. Linking-and-embedding support by itself is quite valuable, but you can take embedded objects one step further by implementing a powerful user-interface model called in-place activation, otherwise known as Visual Editing, covered in Section Four.

I encourage you to start by reading Chapter 1 to become familiar with all the OLE 2 technologies and how they fit together. Chapter 1 also details the information that is in all the later chapters. From there, set your goal and map out an approach that helps you first build a foundation on which you build basic structures like walls and roofing. After that, you can add all the finishing touches on the inside, all the time keeping dry when the monsoons come.

Kraig Brockschmidt
Redmond, Washington
September, 1993

Using the Companion Disks

Bound into the back of this book are two 3.5-inch, 1.44-megabyte companion disks containing the source code files for all the sample programs described in the book. The files on the floppy disks are compressed and must be expanded and copied to your hard disk before you can use them.

To copy these programs to your hard disk, place your disk in the A drive of your computer and enter the following at the MS-DOS prompt:

```
A:INSTALL
```

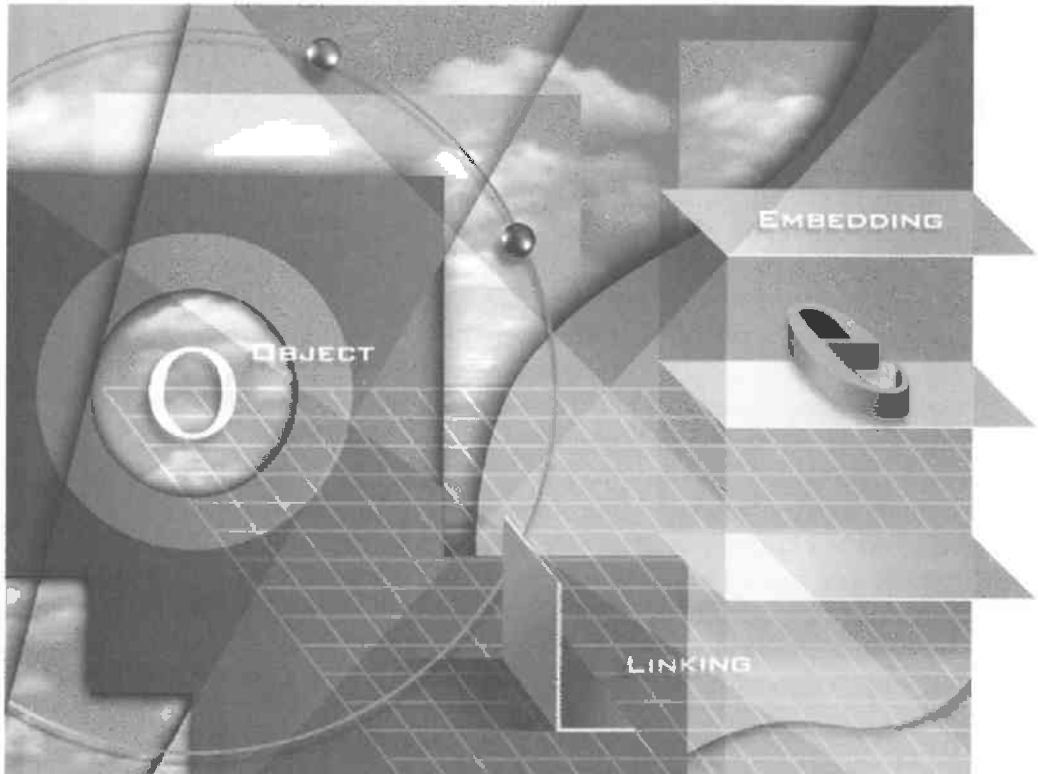
The decompression program on the disk will begin, and you will be told what to do at each step of the unpacking process until all files are copied to the hard disk.

You will need at least 6 MB of space on your hard disk for the sample code that accompanies this book.

Of course, you'll need more than 6 MB of free disk space. The MAKEALL.BAT file in the installation directory will automatically build all the samples for all chapters. The default debug build will consume approximately 80 MB of hard disk space. A "retail" build (type *SET RETAIL=1* at the MS-DOS prompt before running MAKEALL.BAT) will consume considerably less disk space but still requires approximately 50 MB or so. Your compiler is likely to need a megabyte or two extra for temporary files it creates along the way, so be sure you have enough room.

SECTION I

WINDOWS OBJECTS



C H A P T E R O N E

AN OVERVIEW OF OLE 2

All evolution in thought and conduct must at first appear as heresy and misconduct.

—George Bernard Shaw (1856–1950)

Many years from now, a Charles Darwin of computerdom might look back and wonder how the Microsoft Windows APIs (Application Programming Interfaces) evolved into Windows Objects, an object-oriented operating system. OLE version 2 is the genesis of this transformation—it will change how you program—and eventually how you use—Windows. In the beginning, you'll probably regard it as utterly strange and difficult, no matter what your background. But don't feel too threatened. I won't ask you to throw away any knowledge you've accumulated. Instead, we'll ease into the features of OLE 2 and see how those features, combined with everything you already know, can help you reach new heights in your applications.

Today, Windows' features are exposed to applications through a large—and growing—collection of randomly named API functions. (Remember when you first learned that *DeleteObject* is the opposite of *CreateBrush*?) Every API function is created equal, so to speak, and is accessible from virtually any piece of code, regardless of how useful such access really is. Over the years, many new API functions have emerged, each in its own way describing some new capability of the system, each in its own way providing yet another different set of functions by which an application implements various features, and each with its own naming convention (or lack thereof).

Such an environment is a ripe opportunity for object-oriented tools to flourish. Languages such as C++ and class libraries such as Microsoft's Foundation Classes or Borland's Object Windows Library provide some order in the chaos of the API waters. For example, instead of dealing with a window by means of a handle and numerous API functions spread thinly through the reference manuals, these products shelter a window handle

in a C++ object class and directly provide member functions to manipulate the window by means of the object instead of the handle.

In addition, because a member function is always called by means of an object variable, for convenience the names of those functions are located together in the reference manuals, categorized by the object name itself.

In the same manner—and independent of the programming language you choose—OLE 2 exposes system features through what are called "Windows Objects" instead of through API functions. Basically, a Windows Object is a piece of code that exposes its functions through one or more distinct groups of functions. Each group is called an interface. This arrangement provides much-needed order at the system level: Instead of working with disparate handle-based functions, you work with tightly organized system objects. The object model that describes Windows Objects not only describes how the system exposes its functionality to applications but also how applications expose their functionality to the system and to other applications. Realize, too, that the way in which you *expose* an object does not restrict the way in which you can *implement* an object. As we'll see, C++ is the most convenient language in which to express a Windows Object, but you can use other languages just as effectively.

Windows Objects are built on a foundation that also allows an object's code to live anywhere: within a particular application, in a DLL loaded into an application's task, in another application, or even on another machine (in the future, when OLE is network enabled). The object model that OLE 2 introduces lays the evolutionary groundwork for distributed object computing in the years ahead.

OLE 2 exposes a number of key system features, such as the clipboard and the file system, through specific objects. These objects are implemented on top of the existing Windows API functions such as *SetClipboardData* and *OpenFile*. Using these objects today will, of course, cause a decrease in overall performance because you add another layer of function calls to accomplish the same task. For the programmer, however, the overall surface area of API functions is markedly reduced; many of those API functions are moved into member functions of a particular object that you see only when you are manipulating that object. The only globally accessible API functions that remain are a few that initially obtain a pointer to one of the system objects.

Although you will suffer from a performance penalty today, the object implementations of system features will, I believe, gradually become the native expression of those features. The API functions will still be available, but they will be implemented on top of the objects, transferring the performance penalty to those applications that still use the old API functions.

Eventually, the API functions will be provided as some sort of compatibility layer that exists only for the ability to run old applications. All new system features will be provided exclusively by means of objects. Only those applications that have made the transition to using these system objects will be able to benefit from the newest and most powerful features.

This chapter will introduce each specific feature (or technology) of OLE 2, describing briefly how your application might take advantage of it (that is, profit from it) today. By using these features today, you begin to transform your application to more readily take advantage of Windows' future evolution (that is, profit from it tomorrow). The stick that goes along with this proverbial carrot is that you must read this book.

Our latter-day Charles Darwin will then have plenty to say about the origin of a new species of incredibly sophisticated and powerful applications for Windows.

Windows Objects: The Component Object Model

Windows version 1 had about 350 API functions. OLE 2 has over 100. So by measures of new functionality, OLE 2 is roughly one-third of an operating system. By the measure of its impact on your applications, it has appeal as an entire system in itself. It presents as system objects a number of key operating system features such as memory allocation, file management, and data transfer. The huge number of additional features and functions in OLE 2 can be overwhelming. The first step in adopting these new and powerful technologies is to realize that one doesn't learn and exploit a new operating system overnight—there are a few fundamental concepts to learn first. In addition, many higher-level features of any system build on the lower-level features, and OLE 2 is no different. In fact, OLE 2 makes great use of the idea, as shown in Figure 1-1 on the following page.

The first feature is the Component Object Model, which is partly a specification (hence "*Model*") and partly an implementation (contained in COMPOBJ.DLL provided with the OLE 2 SDK). The specification part results from defining a binary standard for object implementation that is independent of the programming language you decide to use. Objects adhering to this standard earn the right to be called Windows Objects. This binary standard enables two applications to communicate through object-oriented interfaces without requiring either to know anything about the other's implementation. For example, you might implement a Windows Object in C++ that supports an interface through which some other code (the user of that object) can learn the names of functions that can be invoked on that object. The

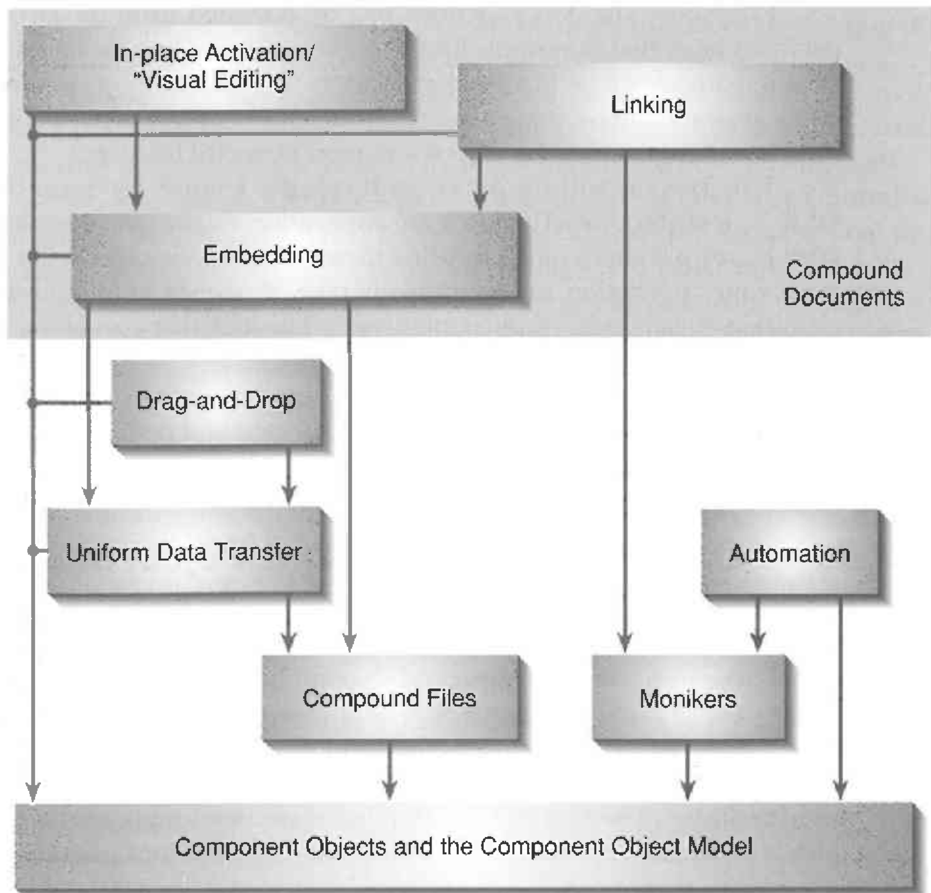


Figure 1-1.
Each feature in OLE 2 builds on lower-level features.

user of this object might be a programming environment such as Visual Basic, or it might be another application written in C, Pascal, Smalltalk, or another language.

The implementation in the component object library (COMPOBJ.DLL) provides a small number of fundamental API functions that allow you to instantiate what is called a *Component Object*, a special type of Windows Object that is identified with a unique class identifier. In return, you are given a pointer to a table of functions (called an interface) that the object implements and through which you can call those functions. This mechanism creates a standard object-creation technique within the system that is independent of the programming language. In addition, this mechanism isolates

you from where the actual object is implemented, which could be in a DLL or another EXE. However, you are oblivious to the location because the component object library handles the communication between modules. In the future, the object might live and execute on another machine on your network, an arrangement that would open the way for distributed object architectures under Windows. Although OLE 2 itself does not contain this feature, it has all the necessary mechanisms into which distributed computing will easily fit.

A Windows Object does not always need to be structured as a Component Object in such a way that the API functions in COMPOBJ.DLL can instantiate it. Use of such API functions is merely one way through which you can obtain your first interface pointer to an object.

There are, of course, other API functions and routes in OLE 2 through which you can obtain that first pointer as well—many of the chapters in this book describe how you generally obtain and use a pointer to specific kinds of objects. When implementing an object, how you allow others to get at your object affects your overall code structure. To make objects addressable via the COMPOBJ.DLL API functions, you must "house" them inside either a DLL (dynamic link library) or an EXE (executable) with specific code—that is, specific functions you call and export from your module. The object itself, however, can be independent of the housing, a capability we will explore in Chapter 4.

The other key piece of implementation in COMPOBJ.DLL handles a process called *marshaling*, or passing function calls and parameters across process boundaries. Because an object's code can execute in another process space and eventually on another machine, COMPOBJ.DLL handles translation of calling conventions and 16-bit to 32-bit parameter translation when the object and that object's user are running in different process spaces. For example, an object might be executing in a 32-bit process space, so it treats types such as UINT as 32-bit values. The user of that object might be running in a 16-bit process space and might call a function in the object passing a 16-bit UINT. In the middle sits COMPOBJ.DLL to marshal that UINT from a 16-bit world into a 32-bit world. Other types, such as pointers, memory handles, and so on, are handled in a similar manner: COMPOBJ.DLL makes sure that each side, object and user, sees the other in terms of its own process space. In the future, when the object can execute on another machine, COMPOBJ.DLL will also account for considerations such as byte ordering.

The need for marshaling is not new: OLE 1 also had to move parameters and memory across process boundaries using Dynamic Data Exchange (DDE). A major problem of OLE 1 that resulted from the asynchronous DDE

protocol was that a function call made on an object was inherently asynchronous, forcing the caller to sit and wait in a message loop until that function was complete, with all the associated problems of time-outs, error recovery, and blocking other requests on the same object. The marshaling mechanism in OLE 2, Lightweight[1] Remote Procedure Call (LRPC), is inherently synchronous—that is, calls made on objects don't return until completed—simplifying the programming model. Some calls, such as those dealing with event notification, remain asynchronous due to the general uses of those calls.

Objects and Interfaces

A technique that describes a binary standard for objects, such as the Component Object Model, does require some change in typical understanding of what the term *object* really means. *Object* is probably the most overused and ambiguous term in the computer industry. *Object* is used everywhere and often with wildly different meanings; as used in this book the term has a specific meaning. Chapter 3 describes a Windows Object in detail, showing exactly how to implement one in both C and C++. Later chapters illustrate a number of routes by which you can obtain a pointer to a specific type of Windows Object. I warn C++ programmers now that a Windows Object is a little different from a C++ object, although you can effectively use C++ objects to implement Windows Objects.

Another term that requires some explanation here is *interface*, another hackneyed and ambiguous term. The notion of interface that applies throughout this book is defined as "a set of semantically related functions implemented on an object." The word *interface* by itself means the *definition* (or *prototype* or *signatures*) of those functions: The OLE 2 include files contain these definitions. An *instantiation* of what I call an *interface implementation* (because the defined interfaces themselves cannot be instantiated without implementation) is simply an array of pointers to functions. Any code that has access to that array—that is, a pointer through which you can get to the top of the array—can call the functions in that interface, as shown in Figure 1-2. Note that in reality, a pointer to an interface is actually a pointer to a pointer to the function table, but that is a detail we can leave until Chapter 3. Conceptually, however, an interface pointer can be viewed simply as a pointer to a function table in which you can call those functions by dereferencing them by means of the interface pointer.

1. "Lightweight" means "no network"; all calls are made on one machine.

The interface definition allows that code to call functions by name and provides type checking on parameters instead of calling functions by an index into the array. Because it's generally inconvenient to draw function tables in expanded form for every interface, this book and other OLE 2 documentation show each function table as a circle (or a jack) connected to the object, as you can see in Figure 1-2.

A Windows Object implements one or more interfaces—that is, it provides pointers to instantiated function tables for each supported interface. A simple object, such as a data object we'll implement in Chapter 6, supports only one specific interface describing data operations such as *GetData* and *SetData*. More complex objects, such as the compound document objects we'll

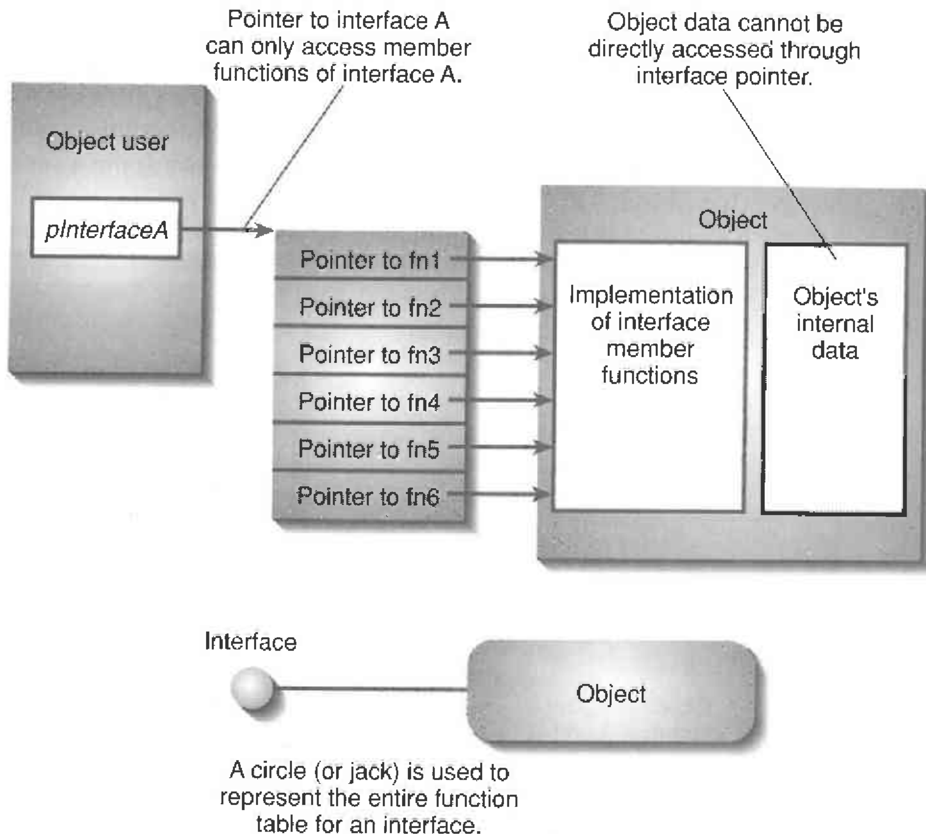


Figure 1-2.
An instantiation of an interface is simply an array of function pointers. A circle (or jack) is a more convenient representation of an interface function table.

implement and use in Chapter 9 and beyond, support at least three interfaces, perhaps more, depending on the features that object implements. Overall, an object is completely described by the collection of interfaces it supports because each separate interface provides the essential manipulation API function to a user of that object.

Whenever the user of some object first obtains a pointer to that object, it has a pointer to only *one interface;* the user never obtains a pointer to the entire object. This pointer allows the user to call only the functions in that one interface's function table, as illustrated in Figure 1-3. Through this pointer, the user has no access to any data members of the object nor does it have any direct access to other interface. In other words, data must be manipulated exclusively through the interface functions, and the interface must have a function through which the caller can obtain a pointer to the object's other interfaces within the object.

Although OLE 2 does not define standard interface functions to access data members of the object, it does define a standard function through which the user of one interface on that object can obtain a pointer to another interface on that object. This function is called *QueryInterface,* as shown in Figure 1-4. We'll examine this function in detail in Chapter 3. When the user queries

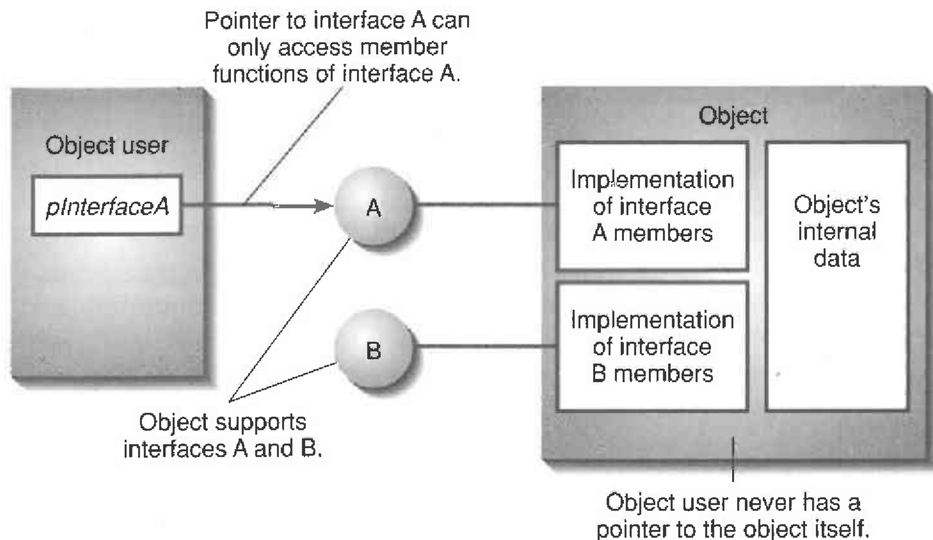


Figure 1-3.
Object users with a pointer to interface A can access only member functions of interface A.

for another interface, it either receives an error (and a NULL pointer), meaning the object does not support the functionality described by the interface, or a valid pointer through which the user might then manipulate the object through that new interface. Because *QueryInterface* is so fundamental, it is part of an interface called *IUnknown* (the *I* stands for *Interface*), which describes the group of fundamental functions that all Windows Objects support, no matter how unknown they are in other respects. All other interfaces in OLE 2 are derived from *IUnknown*, so all interfaces contain the *QueryInterface* function. By implementing one interface on a Windows Object, you automatically implement *IUnknown* because the first few functions in each function table will be those of *IUnknown*, as shown in Figure 1-5 on the following page. (The other two members of *IUnknown* are *AddRef* and *Release*.)

Through *QueryInterface* the user of an object can discover the capabilities of that object at run-time by asking for pointers to specific interfaces. By returning a pointer to that interface, the object is contractually obliged to support the behavior specified for that interface. This enables every object to implement as many interfaces as it wants, so that when it meets a user that

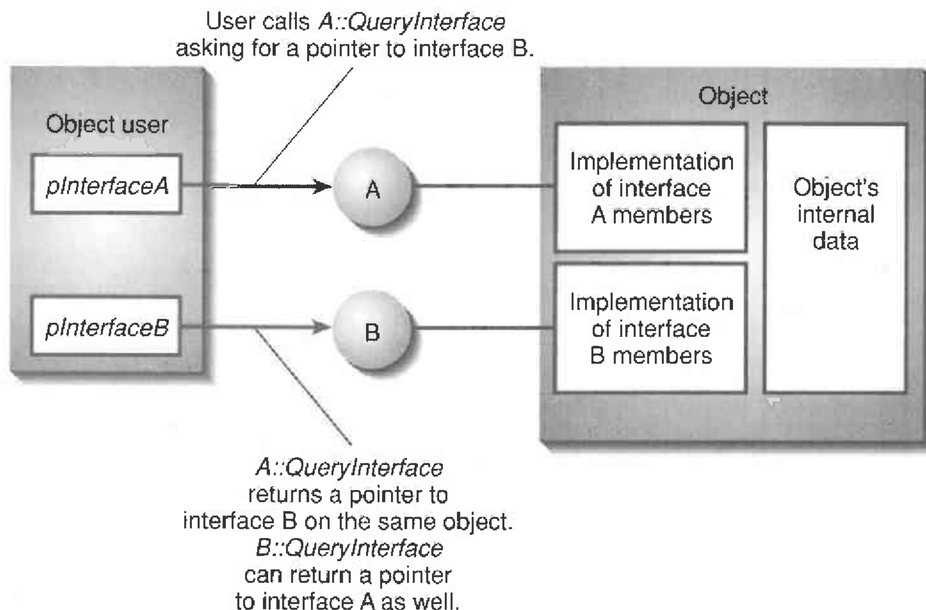


Figure 1-4.
An object user asks the QueryInterface *member of any interface to retrieve pointers to other interfaces on the same object.*

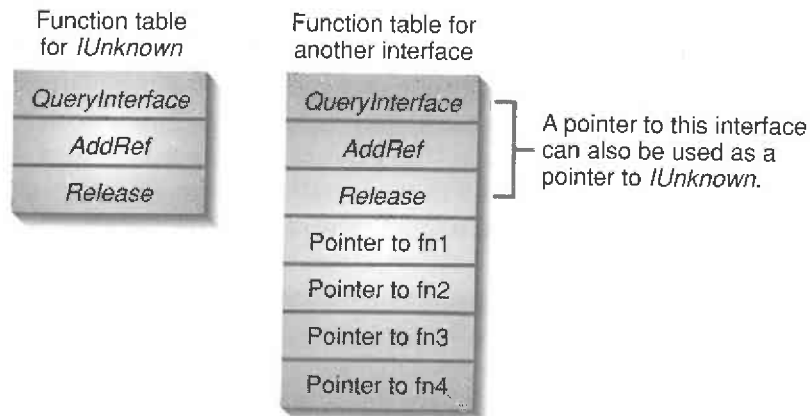


Figure 1-5.
The first few members of any interface are always IUnknown *members. Any interface is therefore polymordial with* IUnknown.

knows how to use many of those interfaces, the two can communicate on a high level. When the object meets a user that knows fewer interfaces, the two can still communicate through the common set of interfaces they both understand—that is, if an object implements interfaces A, B, and C, but the user only knows how to make use of interface B, the object and user can still communicate, but only through interface B. Because all Windows Objects implement at least *IUnknown*, there is always some rudimentary form of possible dialog.

Although OLE 2 defines a large number of standard interfaces, you are free to define and publish your own custom interfaces without requiring any changes whatsoever to the OLE 2 DLLs or any other part of the Windows operating system. The only complication is that you must also provide a DLL for marshaling support because OLE 2's marshaling knows only its own interfaces. But that is a small price to pay for the ability essentially to publish your own new API without having to wait for a system revision from Microsoft.

Structured Storage and Compound Files

The OLE 2 specification defines a number of storage-related interfaces, collectively called Structured Storage. By definition of the term *interface*, these interfaces carry no implementation. They describe a way to create a "file system within a file," and they provide some extremely powerful features for applications. Instead of requiring that a large contiguous sequence of bytes

on the disk be manipulated through a single file handle with a single seek pointer, Structured Storage describes how to treat a single file-system entity as a structured collection of two types of objects—storages and streams—that act like directories and files, respectively.

A stream object is the conceptual equivalent of a single disk file as we understand disk files today. Streams are the basic file-system component in which data lives, and each stream in itself has access rights and a single seek pointer. Streams are named by using a text string (up to 31 characters in OLE 2) and can contain any internal structure you desire.

A storage object is the conceptual equivalent of a directory. Each storage, like a directory, can contain any number of storages (subdirectories) and any number of streams (files), as shown in Figure 1-6. In turn, each substorage can contain any number of storages and streams, until your disk is full.

A storage object does not contain any user-defined data—just as a file-system directory cannot—because it maintains only information related to the storage structure—information about the other streams and substorages that live below it. Each storage has its own access rights—as do streams, a feature that is lacking in MS-DOS directories. Given a storage, you can ask it to

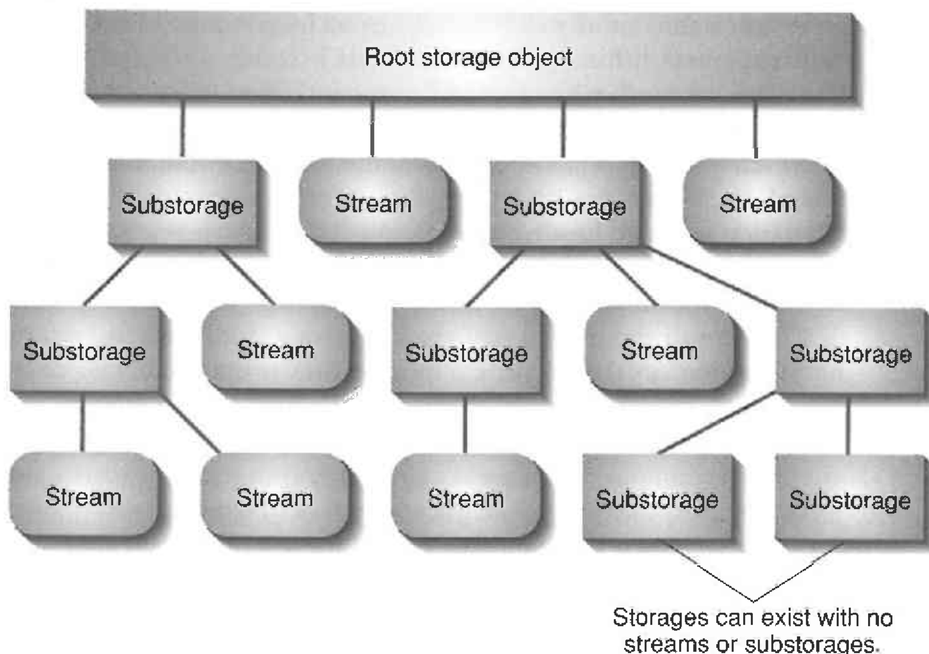


Figure 1-6.
Conceptual structure of storage and stream objects in a compound file.

enumerate, copy, move, rename, delete, or change dates and times of elements within it, providing more than simply the equivalents of MS-DOS commands.

Because Structured Storage is only a specification, OLE 2 provides a complete implementation called Compound Files,[2] which you can use to replace a traditional file handle–based API functions such as *_lread* and *_lwrite*. Do not think that the word *compound* as used here means that compound files are useful only to compound document implementations: The compound files technology is completely independent in the OLE 2 package. In fact, it lives independently in STORAGE.DLL and requires only COMPOBJ.DLL to operate. A similar and 100 percent compatible implementation of compound files will also become the native file system in future versions of Windows, and so, as basic technology, it cannot be restricted to high-level integration features such as the Compound Document standard.

Compound files isolate your application from the exact placement of bytes within your file, just as MS-DOS isolates applications from the exact sectors on the hard disk that your file occupies. MS-DOS presents disparate sectors as a contiguous byte array when you access that file by means of a file handle. In the same manner, compound files present information in a stream as one contiguous entity although the exact information in that stream might be fragmented within the actual file itself.

This means that by adopting compound files for your storage, the physical layout of your files on the disk will no longer be under your direct control. However, although you lose control of the physical layout, you still retain control of which data structures are written into which streams within the file. If you don't want to hassle with reorganizing your structures, you can create a compound file with a single stream, where the stream contains the same structure as your existing file format.

Microsoft recognized that changing your on-disk file format might not be an option, so the use of compound files is optional. The only kind of application that is required to use some aspect of this storage model is a compound document container, which must provide a storage object to any contained compound document object. However, you can create a storage object in memory and later write the contents of that memory into your own file format, as detailed in Chapters 5 and 9. Storage objects created on different storage devices, such as memory and disk files, are indistinguishable from one another to the user of those objects.

Aside from future considerations, compound files provide a number of key features that you can use today to make a more powerful application. For

2. Formerly called *DOCFILES*.

example, you could add additional features yourself that would otherwise be too difficult or time-consuming, such as transactioning and incremental saves. Chapter 5 discusses all the features of compound files and demonstrates both simple and complex uses of this technology. All the features can greatly improve an application's design and treatment of storage.

Structured Storage, as well as compound files, is important for a number of reasons, not the least of which is to standardize the layout of pieces of information within a file. Such standardization enables any piece of code, be it the system shell or another application, to examine the structure of the entire compound file. The exact data formats of each individual stream is still private to whatever wrote that data, but anyone can look into a compound file and enumerate the storages and streams it contains. The OLE 2 Software Development Kit (SDK) even contains a tool called DFVIEW.EXE that displays the structure of any compound file and allows you to dump the hex data of any stream.

Further standardization of the names and contents of a few specific streams (but by no means all streams) enables the system shell and other applications to allow end users to search for occurrences of data within files that match attributes such as creation date, author, keywords, and so on. Microsoft is determined to work with other independent software vendors (ISVs) to define standard names and structures for streams that contain information useful in such queries. The long-range goal is to have all information on the file system structured in such a way that end users can browse the contents of many streams using the system shell. This capability is far more powerful, yet easier to use, than requiring the end user to first find a file, then find the application that can load that file, and then use the application to open and browse files to eventually find the data. Structured Storage enables shell-level document searching, an important manifestation of Microsoft's *Information At Your Fingertips* philosophy.

Uniform Data Transfer and Notification

Built on top of both the Component Object Model and the Compound Files technology is a technology in OLE 2 called Uniform Data Transfer, which provides the functionality to represent all data transfers—clipboard, drag-and-drop, DDE, and OLE—through a single piece of code called a *data object*. Such data objects are not restricted to transferring data through global memory either—they can use other mediums such as compound files. In general, a data source can choose the *best* method for data exchange—that is, the most efficient format and medium of transport. End users benefit from better

performance. Add that to direct, streamlined capabilities such as drag-and-drop and you have a more usable environment overall.

Up to now, all data transfer between an application and anything external (for example, clipboard, drag-and-drop, DDE, or OLE 1) has used global memory. The specific data format contained in that global memory was described by using a clipboard format such as CF_TEXT or CF_BITMAP. Windows (not to mention the programmer) has suffered immensely from inherent limitations of global memory transfers as well as from having radically different protocols and unrelated API functions for exchanging data via clipboard, drag-and-drop, DDE, and OLE 1.

OLE 2 makes two major improvements. First, it allows you to describe data using not only a clipboard format, but also a specification about how much detail the data contains, what type of device (primarily printers) it was rendered for, and what sort of medium is used to transfer the data. This new method of describing and exchanging data, which we'll examine in Chapter 6, is much more powerful than anything previously available. Instead of simply saying "I have a DIB," I can say "I have a thumbnail sketch of a DIB rendered for a 300 dots per inch (dpi) PostScript printer, and it lives in a storage object." For a source of data, you can choose the best possible medium in which to transfer data, and you can make it the preferred format, providing other mediums as backups (such as global memory, the lowest common denominator). So if you happen to generate 30-MB 24-bit DIBs, you can keep those in disk files or storage objects, even during a data exchange. You don't have to load that entire DIB into memory simply for such a transfer.

Data transfer in OLE 2, therefore, can use a compound file, disk file, global memory, or whatever medium is most preferable for data. Understanding that data transfer works on top of compound files, you can see how this OLE 2 feature builds on a lower feature, much in the way that today's clipboard takes advantage of Windows' kernel memory allocation primitives.

Secondly, OLE 2 separates the means of setting up a data exchange—the protocol—from the actual operation of exchanging data. The problem today is that the four transfer protocols (clipboard, File Manager drag-and-drop, DDE, and OLE 1) use widely different functions and widely different data structures, and each has its own limitations. Under OLE 2, applications use new API functions to transfer a *pointer* to a data object from the data source to the consumer of that data. These API functions form the protocol, as discussed in Chapters 6 through 8. After this pointer has been exchanged, the protocol disappears, and all exchange of data happens through the data object. In other words, the protocol worries about exchanging a data object; the data object standardizes how to exchange data rendered in some medium

independent from the protocol. Because the data object does not know anything about protocols, you can write one piece of code to perform an operation such as Paste regardless of how you obtained the data object, hence the "Uniform" in Uniform Data Transfer.

Notification

Consumers of data from an external source are generally interested in when that data changes. OLE 2 handles notifications of this kind through an object called an *advise sink*—that is, a body that absorbs asynchronous notifications from a source. The advise sink not only handles notifications for data changes, but it also is generally used to detect changes in another compound document object, such as when it's saved, closed, or renamed. We'll first see advise sinks in Chapter 6, and we'll see them again in Chapter 9 and beyond.

Data Objects and the Clipboard

Applications can first make use of data objects for Cut and Copy clipboard operations. As Chapter 7 shows, a data object is programmatically similar to common clipboard-handling code. When your data object renders data, you use the same functions you used to generate a handle to pass to *SetClipboard-Data*. When your data object is asked to enumerate the formats it supports, it does so in the same order your clipboard code always has. In fact, a data object used for Copy and Cut operations can be implemented on top of whatever clipboard-handling code you currently have, with some minor modifications, primarily to handle delayed rendering if you have not already.

Pasting data from the clipboard is a matter of retrieving a data object that describes what data is currently on the clipboard. Instead of asking about availability with *IsClipboardFormatAvailable*, you ask such a data object whether it can render a specific format for whatever device, content, and transfer media you want. If the data object can provide the data, you can, at any time, ask for a rendering through the object instead of through *GetClipboardData*.

Data Objects and Drag-and-Drop

Converting an application so that it uses data objects for clipboard transfers is not much of a benefit in and of itself. However, after you have the data object implemented for the clipboard, you can use that same implementation for drag-and-drop. OLE 2 does not deal with the simplistic drag-and-drop of files from File Manager: OLE 2 provides for *full* drag-and-drop of *any* data that you could transfer through the clipboard. Instead of being limited to files or maybe simply to compound document objects, you can write your application to drag and drop any data that you can describe in a data object.

Think of drag-and-drop as a streamlining your existing clipboard operations by eliminating menus, allowing direct manipulation, and providing dynamic feedback to the user about what data is being dragged and what might happen if the data is dropped. In this model, the source of the drag provides the data object, determines what starts and stops the operation, and controls the mouse-cursor–related user interface. The target of a drag receives the data object, checks for usable formats, and determines what will happen with the data if it's dropped inside the target window.

Drag-and-drop is a tremendous user benefit, and if you implement a data object for the clipboard first, your drag-and-drop implementation is close to trivial, as we'll see in Chapter 8: An implementation will not take more than a few days, depending on how fancy you want to get. For the simplest implementation of a drop source you can copy code straight from this book and probably have it working in under an hour. Targets are a little more complicated, but simple targets could be written in an afternoon. You won't find another feature this powerful and this easy to implement.

Data Objects and Compound Documents

Drag-and-drop is not the end; implementing linking and embedding (what we call Compound Document technology) involves augmenting the data object to handle OLE 2 formats to describe both linked and embedded objects. You will modify your data object code to enumerate and render a few new formats; most of the rendering can be delegated to functions already implemented in the OLE 2 SDK's sample code. We'll examine how the Compound Document technology affects data transfers in various ways in Chapter 9 and beyond.

After you have augmented the data object for OLE 2 formats, you instantly enable transfers of Compound Document objects via clipboard and drag-and-drop because neither mechanism cares what the data object actually contains. In addition, by providing OLE 2 clipboard formats in a data object, OLE automatically generates OLE 1 formats for backward compatibility. With an OLE 2 application, you get such backward compatibility for free by simple virtue of using a data object.

If you are familiar with OLE 1, you need to be aware that exchange of an object's native data is now handled through a storage object that represents the part of the container's compound file that is set aside for the object. In OLE 1, the object was asked to allocate global memory and copy its native data into it. In OLE 2, the object instead is given the storage object pointer through which it writes its native data as if it were writing to a file, resulting in much better performance.

Data Objects and DDE

OLE 2 itself doesn't attempt to address data transfers with DDE by use of data objects, for reasons outlined in Chapter 6. It is possible to design a protocol that you could use to isolate your application from DDE and treat it, again, with a data object just as you would treat any other data transfer. Although such a design is outside the scope of this book, it would allow us to come full circle, supporting the four data exchange mechanisms in Windows by means of data objects, and keeping different protocols to retrieve a data object, but treating that data object uniformly from that point onward.

Compound Documents: Object Embedding

The Component Object Model, Compound File, Uniform Data Transfer, and Drag-and-Drop technologies constitute the bulk of OLE 2 that is not concerned with creating applications to support compound documents. The rest of OLE 2 supports what is known as *linking and embedding*. The Compound Document technology is now only a subset of the OLE 2 functionality[3] which builds on the lower-level technologies, as illustrated previously in Figure 1-1 on page 6. The Compound Document technology is first and foremost a standard for integration between applications that follows the standards provided in the lower layers: the Component Object Model standardizes how an object and object user communicate; compound files standardize file structure; Uniform Data Transfer standardizes data exchange functions.

A compound document is essentially a collection site for data from a variety of other sources (that is, other applications). A word processor document, for example, might contain a chart, a table, a metafile drawing, and a bitmap, all of which were created in different applications. Before Object Linking and Embedding, you created such documents by creating the data in another application, copying it to the clipboard, and then pasting it into the document as what we can generically (and rather loathingly) call an object. The clipboard works very well for creating objects, but it does not work so well for later making modifications to those objects because the pasted data no longer retains any information about the application that created it, and it does not retain any of the native data structures that the application used to create it. So when end users want to modify an object, they first have to remember which application was used to create the object, manually locate and launch that application, and then attempt to copy and paste the object back into that application for editing. Because almost all of the native data

3. A historical note: OLE 1 was concerned only with compound documents and provided no other technologies.

structures used to create that object were lost, what is pasted is not what was originally created or even what was originally in the compound document. In most cases, end users were lucky if they could make this work.

The solution to this before OLE was that the application creating an object—what we call a server—and the application that maintained the compound document—what we call a container—shared some sort of private protocol between them, allowing a higher-fidelity transfer. The problem was that no one wanted to maintain private protocols for every other application on the market, and so there had to be a standard. OLE's Compound Document technology is that standard. The server packages its objects so that they are usable in any container written to understand those packages. Because both applications write to a standard instead of to each other, they can achieve high-fidelity integration without any specific knowledge of the other. They communicate through standard OLE interfaces, which provide for editing (or otherwise manipulating) an object, exchanging an object's data, and storing the object's native data structures somewhere in the compound document itself. Custom interfaces allow two applications to achieve even tighter integration than what OLE itself provides, but the existence of such interfaces does not interfere with the standard interfaces because of the *QueryInterface* mechanism mentioned earlier, in the "Objects and Interfaces" section. For most purposes, OLE's Compound Document technology eliminates most of the need for custom interfaces.

Chapter 9 explores container applications that provide site objects that describe places in which an embedded compound document object can live. These site objects implement at least two interfaces, one of which describes containment functions and another that provides functions through which the container is notified of events in the object. Much of the implementation of a container is user-interface oriented, providing dialog boxes such as Paste Special, Insert Object, and Convert Type (new in OLE 2). Fortunately, various groups at Microsoft have contributed to writing a source code library of these dialog boxes as well as other user interface helper functions that should save you tremendous amounts of time implementing a container.

Chapters 10 and 11 explore compound document objects and how to implement them in either DLL or EXE servers. These chapters deal only with embedded objects, which store their private data structures in a storage object provided by the container. This storage object, which is usually some piece of a larger compound file, is for the object's exclusive use. The object can create any kind of structure within that storage object—that is, as many streams and substorages as you want. When asked to save itself, the object writes into this storage, which is essentially writing directly into the container's file. This

means that the object is the only agent that needs to access that storage, and it has the ability to access only as much as necessary. This is a stark contrast to OLE 1's Compound Document technology, in which the container always had to load the entire object's data from a file and pass it to the object via global memory. Under OLE 2, containers only need to pass a pointer to the storage object, resulting in much better performance than OLE 1 could achieve.

You will notice that by Chapter 9 the flavor of this book changes from discussions about specific interfaces and what you *can* do with them into step-by-step guides to compound document interfaces and what you *must* do with them. This reflects the shift from the lower-level and more generic interfaces to those that specifically deal with the Compound Document standard. Standards require predictability, so the step-by-step guides in these later chapters describe how exactly to implement that standard. This is very important because when we talk about compound documents, we're talking about the interaction between two applications that don't know about each other, and that means a certain degree of conformity must be met to prevent a radical increase in entropy.

Compound Documents: Object Linking and Monikers

Enabling container and server applications for linking is a matter of dealing with an additional OLE 2 data format (describing a *link source*) and adding a few more interfaces to the objects that each application implements. The addition of linking capabilities requires very few changes to other compound document code that you implement to handle embedding. Chapters 12 and 13 deal with the necessary changes to support linking in a container and in an object server, respectively.

Linking in OLE 2 affects containers more than it did in OLE 1. Containers are more than just consumers of linked objects—they can become link sources in and of themselves. OLE 2 provides the mechanisms by which a container can provide link source information for objects embedded within their documents. Within the same container, therefore, you can create an embedded object to which another object is linked. Chapter 13 deals specifically with this.

Linked objects have been a significant difficulty for programmers since the beginning of OLE. Because the linked object's data lives in a separate file in the file system, links are easily broken when the end user manually changes the location of that file. OLE 1 depended on absolute pathnames to linked files, so any change in that file's location broke the link, even when the relative paths between the container and the linked file remained the same. In

addition, OLE 1 could not describe more than one layer of nested objects. To solve most of the link breakage problems, as well as to provide for arbitrarily deep object nestings, OLE 2 introduces a type of object called a *moniker.*

A simple moniker contains a reference to linked data and code that knows how to "bind" that link. Binding means different things for different types of monikers. For instance, binding can mean launching applications, loading files, and requesting pointers to interfaces. The most common use of monikers is to identify the source data for a linked object in a compound document. This generally requires a reference to a file (such as a filename) and an identification of the part of that file that is the actual source of the link. To accommodate this, OLE 2 provides a simple *file moniker* to manage a filename and an *item moniker* to manage some identification for a portion of a file.

A file moniker contains some sort of pathname, which can be as simple as an eight-character filename (with or without a three-character extension) or as complex as a full pathname, drive letter included. A linked object actually maintains two file monikers (and can have other monikers, as well): one with a full pathname to the linked file (such as C:\STATUS\JUNE\REPORT.DOC) and another with a relative pathname (such as ..\JUNE\REPORT.DOC). When a linked object is asked to bind to its source, it first asks the absolute file moniker to bind, which means launching the application that knows how to load the file (based on the extension). It then asks the application for an interface that knows how to load files, and then it asks that interface to load the actual file. If loading fails, the linked object tries to bind the relative moniker, which executes the same process. Although the absolute pathname is the fastest and most reliable way to get to a file, the relative moniker addresses the cases in which an entire directory tree was moved, breaking absolute links but not relative links. The only case that is not handled is when the user moves a source file to a completely new location. That cannot be solved until the operating system is aware of every such change. When that happens, the system will automatically update the link paths without the application knowing or caring.

An item moniker is simply some sort of name that makes sense only to the application that originally created it and provided it as part of a linked object. Binding to an item moniker means asking the application (presumably loaded through a file moniker already) for an interface that knows how to resolve the name into some sort of pointer to an interface on the actual object.

A linked object generally stores complex references in a composite moniker that is a collection or sequence of other simple monikers. Most links are expressed in a composite of one file moniker and one item moniker. Longer sequences of monikers express more complex notions, such as nested

objects in which the composite contains many item monikers. An illustration of a composite moniker that contains a file and an item moniker is shown in Figure 1-7.

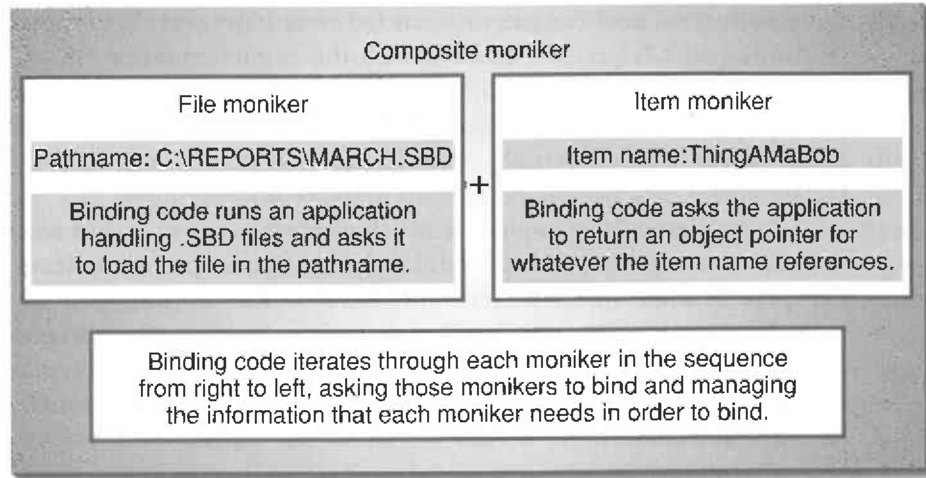


Figure 1-7.
Conceptual file, item, and composite monikers.

Compound Documents: In-Place Activation (Visual Editing™)

The Compound Document technology provides a way to embed an object in or link an object to a container document. The container provides the appropriate functionality to activate the object, which can invoke any number of actions on the object. When the action implies editing, both linked and embedded objects are opened in another window, which provides the editing context, the same model OLE 1 supported.

To stress a document-centric view of computing, OLE 2 provides the ability to *activate* an object in place—inside the container application's window. Editing the object is a subset of the more generic action of activating. Instead of the object opening another window to execute an action, it might choose to provide editing tools or other controls in the context of the container. End users benefit from never having to leave the document context in which they are working—no distractions of other windows and other environments. Instead of seeing two copies of the data, one in the container and one in a separate editing window, end users see only the one in the container, in the full context of its containing document.

Servers, containers, and compound document objects require a number of additional changes and interfaces to support in-place activation, but these build on top of and use much of the compound document implementations that you'll already have done by this time. We'll discuss the mechanisms of in-place activation and the interactions between objects and containers at the beginning of Chapter 15, leading into the implementation of an in-place capable container application. We'll then discuss the implementation of in-place objects in Chapter 16, and we'll end that chapter with an exercise in extrapolation to see where the in-place activation technology might take us in the future.

By implementing in-place activation interfaces, you do not restrict your container or object to being useful only to other in-place applications. The presence of the in-place activation interfaces does not interfere in any way with the more basic compound document interfaces. So if you implement an in-place object, it will be useful to an in-place container, which can activate it in place, as well as to a simpler container, which will always activate the object in a separate window.

Automation

One other key technology that is part of the OLE 2 system but that is completely separate from the rest is OLE Automation. This technology allows an object—any object regardless of other features—to expose a set of commands and functions that some other piece of code can invoke. Each command can take any number of parameters, and automation provides the methods through which an object describes the names and types of those parameters. A full description of this technology, while very rich and exciting, is beyond the scope of this book. In fact, automation is a large enough topic to justify an entire book by itself. These few paragraphs constitute all that I mention in this book about the technology.

The intent of Automation is to enable the creation of system macro programming tools. Such tools will ask automation-enabled objects for lists of their function names and lists of parameters (names and types) that those functions accept. At one point, Microsoft was considering a single system macro programming tool, but this approach would have meant one language and one tool for all end users. Through Automation, OLE 2 allows objects to describe their capabilities to any tool where the tool defines the programming environment. Ultimately this gives the end user the choice of language, vendor, functionality, and so on.

Automation objects generally describe user-level functions on the order of File Open or Format Character; Automation tools display those functions to the user, allowing the user to write macro scripts that span applications.

The major motivation for this mechanism is to pave the way for programming tools that can affect any Automation-enabled object, regardless of whether the system or an application implements that object. When the user chooses an object, such a tool asks the object for its list of function names and exposes to the user the operations that are possible on the object. When the user selects a function to use, the programming tool can further ask the object for the names and types of that function's parameters, and it can provide the environment in which the end user can indicate what to pass in each parameter.

With a system full of such objects from many applications, the end user can use any programming tool that understands Automation to write macros that *could* span applications. A more immediate benefit to your specific application is that such a tool can also write macros that operate only in your application, eliminating the need for you to create your own specific macro language. The end user is then free to choose his or her preferred programming tool, any of which use your Automation interfaces in the same way. The user gains the benefit of having one tool that works with all Automation applications; you benefit from exposing Automation once and letting someone else provide the programming environment.

But it doesn't stop there. Although Automation exposes commands through which external agents invoke your functions, there is no reason whatsoever that you could not invoke those same commands yourself. You might implement Automation on top of your application's message procedure, or you might choose to implement your message procedure on top of Automation. Looking ahead, centralizing such code in the Automation interface might lead to the eventual elimination of message procedures, using instead a more general-purpose and powerful command processing object.

CHAPTER TWO

CONVENTIONS, C++, AND SAMPLE CODE

Throughout this book, we'll watch two applications evolve as we learn how applications can take advantage of the various OLE 2 technologies and what pieces of code are necessary to achieve the apex of in-place activation, which will doubtless be the goal of many readers.

One application will be written from scratch—that is, it will not implement certain features such as file I/O until we can use compound files. This application is suitable to become a container application, but before that time, it will serve to illustrate how to incorporate non-compound-document features.

The other sample is a full-featured application that without doing anything else terribly important uses the traditional Windows API functions from the start, implementing a number of features common to all applications, such as clipboard exchanges and file I/O. As we follow this application through each chapter, we'll replace the use of Windows API functions with the use of OLE 2 technologies, such as converting existing file I/O into compound files. Beginning in Chapter 4, we'll break a piece of this application into a component object DLL and separately develop it into an embedded object capable of in-place activation.

Along the way, we'll also create a number of useful components (either code fragments or DLLs) that you might find helpful in your own implementations.

To C or Not to C (with Apologies to Shakespeare)

The sample code provided in this book is mostly in C++, primarily because the concepts and features of OLE 2 are best expressed in that language. Authoring a book of this sort presents a few philosophical difficulties, such as

what language to use, how everything will fit on the companion disks, and how not to alienate a large portion of your audience.

C++ code is smaller and simplifies code reuse, reducing the amount of code I have to write and the amount of code you have to read. C programmers will no doubt be a little put off by this, so in this section I've provided critical explanations of basic C++ concepts and notations that should help the C programmer understand the sample code. While writing the code, I tried to remember that it has to be understandable to a typical C programmer, so I've purposely kept myself from going hog wild about everything C++ can do, such as deep multiple inheritance or long chains of virtual functions. This will no doubt put off a number of C++ programmers, but believe me, it is not as bad as forcing everyone to labor through verbose C.

Another possible source of irritation is that I wrote these samples in C++ using my own class library (called CLASSLIB) instead of a real library such as the Microsoft Foundation Classes, which you might be using and for which you might harbor a religious zeal. The reason is that libraries such as Microsoft Foundation Classes, although very convenient, tend to hide much of what we need to discuss and have a strong tendency to render an application utterly foreign to C programmers, who develop glazed expressions and start asking questions such as "Where is *WinMain*?" and "Where's the window procedure?" All the samples in this book have a *WinMain* (or a *LibMain*) from which you can follow the thread of execution. The class libraries I wrote for this book serve mostly to keep a lot of the basic code for a Windows program out of the way, and they were something I could include on the sample disks.

With the exception of the code to manage the application's data structures, the sample code was originally written in straight C. In fact, these applications were ported from original C versions mostly by changing structures into classes, which represents nearly the extent of my C++ talents. A C programmer briefed on the fundamental rules of C++ should be capable of taking the classes back to structures mostly by means of global search and replace instead of a line-by-line rewrite.

The remainder of this section is intended to be a C++ briefing for C programmers, explaining this newer language from a C perspective so that you can work through the rest of the code in this book. This section does not describe any details about OLE 2 itself but covers the aspects of the C++ language that I used in this book's samples to implement OLE 2 features. (Note that when I use the word *object* in this section I mean a C++ object, not a Windows Object, as I will mean in the rest of the book.) C++ is a matter of convenience and results in much more compact code. I do not claim to be a C++ expert, so please refer to any of the plethora of C++ books available

to make more sense out of this language. If you are already comfortable with your C++ knowledge, feel free to skip to the "Sample Code" section of this chapter, which starts on page 41.

User-Defined Types: C++ Classes

Many a C application is built on top of a number of data structures, one of which might be a typical user-defined structure of application variables such as the following:

```
typedef struct tagAPPVARS
    {
    HINSTANCE   hInst;              //WinMain parameters
    HINSTANCE   hInstPrev;
    LPSTR       pszCmdLine;
    int         nCmdShow;

    HWND        hWnd;               //Main window handle
    } APPVARS;

typedef APPVARS FAR *LPAPPVARS;
```

To manage this structure, an application will implement a function to allocate one of these structures, a function to initialize it, and a function to free it.

```
LPAPPVARS AppVarsPAllocate(HINSTANCE, HINSTANCE, LPSTR, int);
BOOL      AppVarsFInit(LPAPPVARS)
LPAPPVARS AppVarsPFree(LPAPPVARS);
```

When another piece of code wants to obtain one of these structures, it calls *AppVarsPAllocate* to retrieve a pointer. Through that pointer, it can initialize the structure with *AppVarsFInit* (which in this case might attempt to create a window and store it in *hWnd*) or access each field in the structure.

By creating this structure and providing functions that know how to manipulate that structure, you have defined a type. C++ formalizes this commonly used technique into a *class* defined by the *class* keyword:

```
class __far CAppVars
    {
    public:
        HINSTANCE   m_hInst;            //WinMain parameters
        HINSTANCE   m_hInstPrev;
        LPSTR       m_pszCmdLine;
        int         m_nCmdShow;

        HWND        m_hWnd;             //Main window handle

    public:
        CAppVars(HINSTANCE, HINSTANCE, LPSTR, int);
```

(continued)

```
        ~CAppVars(void);
        BOOL FInit(void);
    };

typedef CAppVars FAR *LPCAppVars;
```

The name after *class* can be whatever name you want. Although we could have used APPVARS, paralleling the C structure, the name *CAppVars* conforms to a C++ convention of using mixed-case names for classes prefixed with a *C* for *class*. Another convention in C++ classes, at least around Microsoft, is to name data fields with an *m_* prefix to clearly identify the variable as a member of a class.

When another piece of code wants to use this class, it must instantiate a C++ object of this class. In C terms, *CAppVars* is a structure. To use the structure, you still have to allocate one. In C++, we do not need separate functions to allocate the structure, nor do we use typical memory allocation functions. Instead we use C++'s *new* operator, which allocates an object of this class and returns a pointer to it, as follows:

```
LPCAppVars    pAV;

pAV=new CAppVars(hInst, hInstPrev, pszCmdLine, nCmdShow);
```

Because *CAppVars* was declared as *__far*, *new* allocates far memory and returns a far pointer. If the allocation fails, *new* returns NULL. But this is not the whole story. After the allocation is complete, and before returning, *new* calls the class *constructor* function, which is the funny-looking entry in the following class declaration:

```
    public:
        CAppVars(HINSTANCE, HINSTANCE, LPSTR, int);
```

To implement a constructor, you supply a piece of code in which the function name is *<class>::<class> (<parameter list>)* where *::* means "member function of," as in the following:

```
CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    //Initialize members of the object
    m_hInst=hInst;
    m_hInstPrev=hInstPrev;
    m_pszCmdLine=pszCmdLine;
    m_nCmdShow=nCmdShow;
    }
```

The *::* notation allows different classes to have member functions with identical names because the actual name of the function known internally to the

compiler is a combination of the class name and the member function name. This allows programmers to remove the extra characters from function names that are used in C to identify the structure on which those functions operate.

The constructor, which always has the same name as the class, can take any list of parameters, but unlike a C function, it has no return value because the *new* operator will return whether or not the allocation succeeded. Because the constructor cannot return a value, C++ programmers typically avoid placing code in the constructor that might fail, opting instead for a second function to initialize the object after it has been positively instantiated.

Inside the constructor, as well as inside any other member function of the class, you can directly access the data members in this object instantiation. The *m_* prefix on data members is the common convention used to distinguish their names from other variables, especially since the names of data members often conflict with parameter names.

Implicitly all the members (both data and functions) are dereferenced off a pointer named *this*, which provides the member function with a pointer to the object that's being affected. Accessing a member such as *m_hInst* directly is equivalent to writing *this->m_hInst*; the latter is more verbose, and so it is not used often.

The code that called *new* will have a pointer through which it can access members in the object just as it would access any field in a data structure:

```
UpdateWindow(pAV->m_hWnd);
```

What is special about C++ object pointers is that you can also call the *member functions* defined in the class through that same pointer. In the preceding class declaration, you'll notice that the functions we had defined separately from a structure are pulled into the class itself. Instead of having to call a function and pass a structure pointer, as follows;

```
//C call to a function that operates on a structure pointer
if (!AppFInit(pAV))
    {
    [Other code here]
    }
```

the caller can dereference a member function through the following pointer:

```
//C++ call to an object's member function
if (!pAV->FInit())
    {
    [Other code here]
    }
```

The *FInit* function is implemented with the same :: notation that the constructor uses:

```
CAppVars::FInit(void)
    {
    //Code to register the window class might go here.

    m_hWnd=CreateWindow(...);   //Create the main app window

    if (NULL!=m_hWnd)
        {
        ShowWindow(m_hWnd, m_nCmdShow);
        UpdateWindow(m_hWnd);
        }

    return (NULL!=m_hWnd);
    }
```

Again, because a constructor cannot indicate failure through a return value, C++ programmers typically supply a second initialization function, such as *FInit*, that performs operations that might be prone to failure.

You could, of course, still provide a separate function outside the class that took a pointer to an object and manipulated it in some way. However, a great advantage of using member functions is that you can only call member functions in a class through a pointer to an object of that class. This prevents all sorts of problems when you accidentally pass the wrong pointer to the wrong function, an act that usually brings about some very wrong events.

Finally, when you are finished with this object, you'll want to perform cleanup on the object and free the memory it occupies. Instead of calling a specific function for this purpose, you use C++'s *delete* operator:

```
delete pAV;
```

delete frees the memory allocated by *new*, but before doing so it calls the object's *destructor*, which is that even-funnier-looking function in the class declaration (with the tilde, ~) but which comes with an implementation like any other member function:

```
//In the class
public:
    ~CAppVars(void);

...

//Destructor implementation
CAppVars::~CAppVars(void)
    {
    //Perform any cleanup on the object.
    if (IsWindow(m_hWnd))
        DestroyWindow(m_hWnd);

    return;
    }
```

The destructor has no parameters and no return value because after this function returns, the object is simply *gone*. Therefore, there is no point in telling anyone that something in here worked or failed because there is no longer an object to which such information would apply. The destructor is a great place—in fact, your only chance—to perform final cleanup of any allocations made in the course of this object's lifetime.

Of course, there are many other ways to define classes and to use constructors, destructors, and member functions than I've shown here. However, this reflects how I've implemented all the sample code in this book.

Access Rights

You probably noticed those *public* labels in the class definition or should, by now, be wondering what they're for. In addition to *public*, two variations of *public* can appear anywhere in the class definition: *protected* and *private*.

When a data member or member function is declared under a *public* label, any other piece of code that has a pointer to an object of this class can directly access those members by means of dereferencing, as follows:

```
LPCAppVars     pAV;
HINSTANCE      hInst2;

pAV=new CAppVars(hInst, hPrevInst, pszCmdLine, nCmdShow);

hInst2=pAV->m_hInst;  //Public data member access

if (!pAV->FInit())    //Public member function access
    {
    [Other code here]
    }
```

When data members are marked as *public*, another piece of code is allowed to change that data without the object knowing, as in the following:

```
pAV->m_hInst=NULL;    //Generally NOT a good idea
```

This is a nasty thing to do to some poor object that assumes that *m_hInst* never changes. To prevent such arbitrary access to an object's data members, you would mark such data members as *private* in the class, as follows:

```
class __far CAppVars
    {
    private:
        HINSTANCE   m_hInst;            //WinMain parameters
        HINSTANCE   m_hInstPrev;
        LPSTR       m_pszCmdLine;
        int         m_nCmdShow;
```

(continued)

```
        HWND        m_hWnd;                //Main window handle

    public:
        CAppVars(HINSTANCE, HINSTANCE, LPSTR, int);
        ~CAppVars(void);
        BOOL FInit(void);
    };
```

Now code such as *pAV->hInst=NULL* will fail with a compiler error because the user of the object does not have access to private members of the object. If you want to allow read-only access to a data member, provide a public member function to return that data. If you want to allow write access but would like to validate the data before storing it in the object, provide a public member function to change a data member.

Both data members and member functions can be private. Private member functions can be called only from within the implementation of any other member function. In the absence of any label, *private* is used by default.

If a class wants to provide full access to its private members, it can declare another class or a specific function as a *friend*. Any friend code has as much right to access the object as the object's implementation has. For example, a window procedure for a window created inside an object's initializer is a good case for a friend:

```
class __far CAppVars
    {
    friend LRESULT FAR PASCAL AppWndProc([WndProc parameters]);

    private:
        [Private members accessible in AppWndProc]

    ...
    };
```

Any member declared after a *protected* label is the same as *private* as far as the object implementation or the object's user is concerned. The difference between *private* and *protected* manifests itself in derived classes, which brings us to the subject of inheritance.

Single Inheritance

A key feature of the C++ language is code reusability through a mechanism called *inheritance*—that is, one class can inherit the members and implementation of those members from another class. The inheriting class is called a *derived class;* the class from which the derived class inherits is called a *base class.*

Inheritance is a technique to concentrate code common to a number of other classes in one base class—that is, placing the code in a place where other classes can reuse it. Applications for Windows written in C++ typically have some sort of base class to manage a window, as in the following *CWindow* class:

```
class __far CWindow
    {
    protected:
        HINSTANCE    m_hInst;
        HWND         m_hWnd;

    public:
        CWindow(HINSTANCE);
        ~CWindow(void);

        HWND Window(void);
    };
```

The *CWindow* member function *Window* simply returns *m_hWnd*, allowing read-only access to that member.

If you now want to make a more specific type of window, such as a frame window, you can inherit the members and the implementation from *CWindow* by specifying *CWindow* in the class definition, using a colon to separate the derived class from the base class, as follows:

```
class __far CFrame : public CWindow
    {
//CFrame gets all CWindows variables.
    protected:
        //We can now add more members specific to our class.
        HMENU    m_hMenu;

    public:
        CFrame(HINSTANCE);
        ~CFrame(void);

    //We also get CWindow's Window function.
    };
```

The implementation of *CFrame* can access any member marked *protected* in its base class *CWindow*. However, *CFrame* has no access to *private* members of *CWindow*.

You will also see a strange notation in constructor functions:

```
CFrame::CFrame(HINSTANCE hInst) : CWindow(hInst)
```

This notation means that the *hInst* parameter to the *CFrame* constructor is passed to the constructor of the *CWindow* base class first, before we start executing the *CFrame* constructor.

Code that has a pointer to a *CFrame* object can call *CWindow::Window* through that pointer. The code that executes will be the implementation of *CWindow*. The implementation of *CFrame* can, if it wants, redeclare *Window* in its class and provide a separate implementation that might perform other operations, as follows:

```
class __far CFrame : public CWindow
    {
    ...
    HWND Window(void);
    };

CFrame::Window(void)
    {
    [Other code here]

    return m_hWnd;    //Member inherited from CWindow
    }
```

If a function in a derived class wants to call the implementation in the base class, it explicitly uses the base class's name in the function call. For example, we could write an equivalent *CFrame::Window* as follows:

```
CFrame::Window(void)
    {
    return CWindow::Window();
    }
```

In programming, one often finds it convenient to typecast pointers of various types to a single type that contains the common elements. In C++, you can legally typecast a *CFrame* pointer to a *CWindow* pointer, because *CFrame* looks like a *CWindow*. However, calling a member function through that pointer might not do what you expect, as in the following:

```
CWindow * pWindow;
HWND      hWnd;

pWindow=(CWindow *)new CFrame();    //Legal conversion
hWnd=pWindow->Window();
```

Whose *Window* is called? Because it is calling through a pointer of type *CWindow* *, this code calls *CWindow::Window*, not *CFrame::Window*.

Programmers would like to be able to write a piece of code that knows only about the *CWindow* class but that is also capable of calling the *Window*

member functions of derived class. For example, a call to *pWindow->Window* would call *CFrame::Window* if, in fact, *pWindow* is physically a pointer to a *CFrame*. To accomplish this requires what is known as a *virtual function*.

Virtual Functions and Abstract Base Classes

To solve the typecasting problem described in the previous section, we have to redefine the *CWindow* class to make *Window* a virtual function using the keyword *virtual*, as follows:

```
class __far CWindow
    {
    ...
    virtual HWND Window(void);
    };
```

The *virtual* keyword does not appear in the implementation of *CWindow::Window*.

If *CFrame* wants to override *CWindow::Window*, it then declares the same function in its own class and provides an implementation of *Window*, like this:

```
class __far CFrame : public CWindow
    {
    ...
    virtual HWND Window(void);
    };

CFrame::Window(void)
    {
    [Code that overrides the default behavior of CWindow]
    }
```

Such an override might be useful in a class that hides the fact that it actually contains two windows; the implementation of *Window* would then perhaps return one or the other window handle, depending on some condition.

With *CWindow::Window* declared as *virtual*, the piece of code we saw earlier has a different behavior, as in this:

```
pWindow=(CWindow *)new CFrame();    //Legal conversion
hWnd=pWindow->Window();
```

The compiler, knowing that *CWindow::Window* is virtual, is now responsible for figuring out what type *pWindow* really points to, although the program itself thinks it's a pointer to a *CWindow*. In this code, *pWindow->Window* calls *CFrame::Window*. If *pWindow* really points to a *CWindow*, the same code would call *CWindow::Window* instead.

C++ compilers implement this mechanism by means of a *virtual function table* (sometimes referred to as a *Vtbl*) that lives with each object. The function table of a *CWindow* will contain one pointer to *CWindow::Window*. If *CFrame* overrides the virtual functions in *CWindow*, its table will contain a pointer to *CFrame::Window*. If, however, *CFrame* does *not* override the *Window* function, its table contains a pointer to *CWindow::Window*.

A pointer to any object in certain implementations of C++[1] is really a pointer to a pointer to the object's function table. Whenever the compiler needs to call a member function through an object pointer, it looks in the table to find the appropriate address, as shown in Figure 2-1. So if the virtual

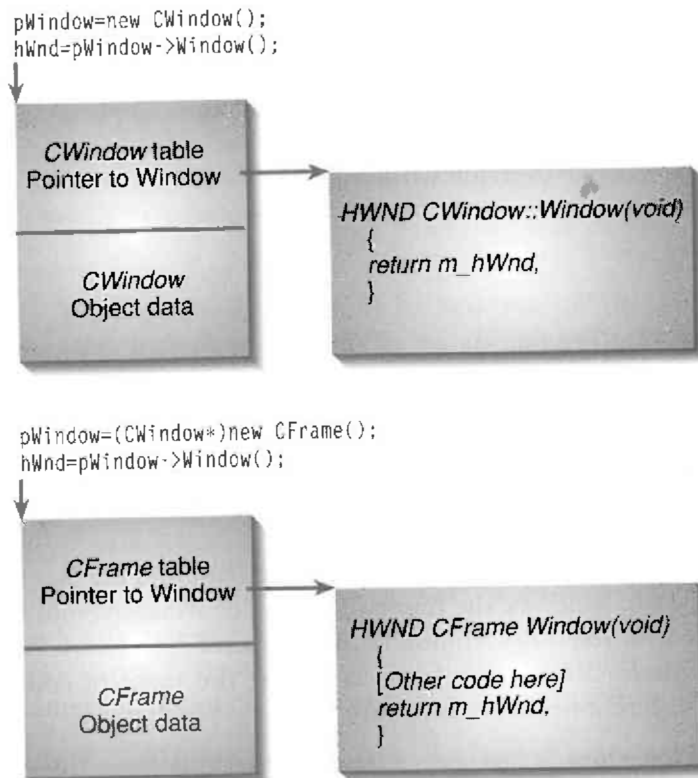


Figure 2-1.
C++ compilers call virtual functions of an object by means of a function table.

1. At least Visual C++ 1.0 and Borland C++ 3.1.

Window of the *CWindow* class and of all derived classes always occupies the first position in the table, calls such as *pWindow->Window* are actually calls to whatever address is in that position.

Virtual functions can also be declared as *pure virtual* by appending *=0* to the function in the class declaration, as follows:

```
class __far CWindow
    {
    ...
    virtual HWND Window(void)=0;
    };
```

Pure virtual means "no implementation defined," which renders *CWindow* into an *abstract base class*—that is, you cannot instantiate a *CWindow* by itself. In other words, pure virtual functions do not create entries in an object's function table, so C++ cannot create an object through which someone might try to make that call. As long as a class has at least one pure virtual member function, it is an abstract base class and cannot be instantiated, a fact compilers will kindly mention.

An abstract base class tells derived classes "You *must* override my pure virtual functions!" A normal base class with normal virtual functions tells derived classes "You *can* override these, *if you really care.*"

You might have noticed by now that an OLE 2 interface is exactly like a C++ function table, and this is intentional. OLE 2's interfaces are defined as abstract base classes, so an object that inherits from an interface must override every interface member function—that is, when implementing an object in C++, you must create a function table for each interface, and because interfaces themselves cannot create a table, you must provide the implementations that will. OLE 2, however, does not require that you use C++ to generate the function table; although C++ compilers naturally create function tables, you can just as easily write explicit C code to do the same.

Multiple Inheritance

The preceding section described single inheritance—that is, inheritance from a single base class. C++ allows a derived class to inherit from multiple base classes and thus to inherit implementations and members from multiple sources. The samples in this book do not use multiple inheritance, although there are no technical reasons preventing them from doing so. They use only single inheritance to remain comprehensible to C programmers who are just beginning to understand the concept. In any case, multiple inheritance is evident in the following class declaration:

```
class __far CBase
    {
    public:
        virtual FunctionA(void);
        virtual FunctionB(void);
        virtual FunctionC(void);
    };

class __far CAbstractBase
    {
    public:
        virtual FunctionD(void)=0;
        virtual FunctionE(void)=0;
        virtual FunctionF(void)=0;
    };

//Note the comma delineating multiple base classes.
class __far CDerived : public CBase, public CAbstractBase
    {
    public:
        virtual FunctionA(void);
        virtual FunctionB(void);
        virtual FunctionC(void);
        virtual FunctionD(void);
        virtual FunctionE(void);
        virtual FunctionF(void);
    };
```

An object of a class using multiple inheritance actually lives with multiple function tables, as shown in Figure 2-2. A pointer to an object of the derived class points to a table that contains all the member functions of all the base classes. If this pointer is typecast to a pointer to one of the derived classes, the pointer actually used will refer to a table for that specific base class. In all cases, the compiler dutifully calls the function in whatever table the pointer referenced.

Of course, there are limitations to using multiple inheritance, primarily when the base classes have member functions with the same names. In such cases, the object can have only one implementation of a given member that is shared between all function tables, just as each function in Figure 2-2 is shared between the base class table and the derived class table.

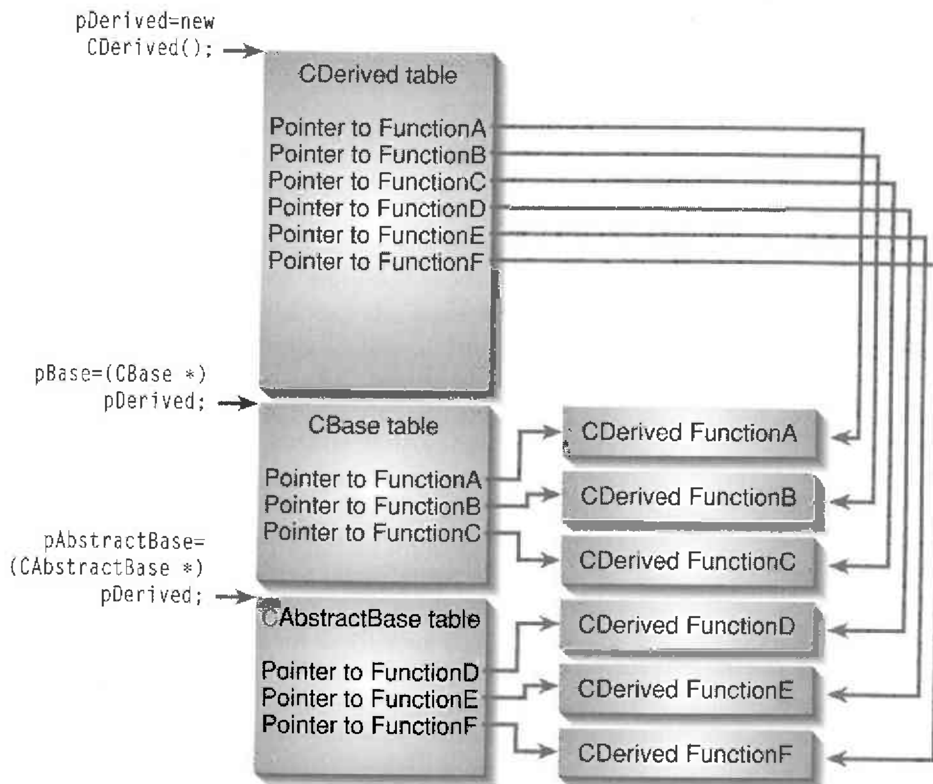


Figure 2-2.
Objects of classes using multiple inheritance contain multiple tables.

Sample Code

In case you have not noticed already, this book contains quite a lot of sample code, enough to require two companion disks. After installing the sample code on your own machine, you will have a number of directories with the contents shown in Table 2-1.

This book follows the development of two applications that you'll find in many CHAP*xx* directories in the sample code: Cosmo and Patron. Both of these applications will compile into single-document or multiple-document versions, depending on the build environment you want. They both make use of a common code base in CLASSLIB, and they use the BTTNCUR, GIZMOBAR, and STASTRIP DLLs to provide user interface components. Most of the sample code depends on the contents of the INC and LIB directories as well, including Cosmo and Patron. To make the purpose of all the code clear, the following sections deal with each directory in detail.

Directory	Contents
INC	Include (.H) files used by more than one sample.
LIB	Libraries (.LIB files) used by more than one sample.
BUILD	A repository for built DLLs and EXEs so that you can include this one directory in your PATH command. Before you build any of the other samples, this directory will contain a build of the OLE2UI library that is shipped with the OLE 2 SDK, customized for the samples in this book (in the file BOOKUI.DLL).
BTTNCUR	Version 1.1 update of the Buttons & Cursors DLL. Compiles into BTTNCUR.DLL.
GIZMOBAR	An implementation of a toolbar, called the GizmoBar, which compiles into GIZMOBAR.DLL.
STASTRIP	An implementation of a status-line control, StatStrip, which compiles into STASTRIP.DLL and includes a small test program in the DEMO directory.
CLASSLIB	A specific C++ class library as used by the more feature-laden samples.
INTERFAC	Template implementations for all the OLE 2 interfaces discussed in this book.
CHAP*xx*	Sample code for Chapter *xx*.

Table 2-1.
Directories created after the companion disks are installed.

Include Files: The INC Directory

The INC directory is a repository for any .H file that is used from more than one application. The files stored in the directory and their use are listed in Table 2-2:

File	Purpose
BOOK1632.H	Macros that isolate the application from Win16 and Win32 differences.
BOOKGUID.H	Definitions of Globally Unique Identifiers (CLSIDs and IIDs) used in all samples in this book, as well as anything else generally useful to all samples, such as OLE 2-related macros.

Table 2-2. *(continued)*
Contents of the INC directory in the sample code.

Table 2-2. *continued*

File	Purpose
BTTNCUR.H	Definitions for BTTNCUR.DLL. Identical to BTTNCUR\BTTNCUR.H.
CLASSLIB.H	Include file for the class library. Identical to CLASSLIB\CLASSLIB.H.
CLASSRES.H	Resource constants for applications using the class library. Identical to CLASSLIB\CLASSRES.H.
DEBUG.H	Macros to facilitate simple debug output.
IENUM0.H	A file shared by both samples in Chapter 3.
IPOLY*x*.H	Definitions for a POLYLINE.DLL from Chapters 4, 5, and 6, where *x* represents the applicable chapter. See the later section titled "Cosmo: A Graphical Editor."
GIZMOBAR.H	Definitions for GIZMOBAR.DLL. Identical to GIZMOBAR\GIZMOBAR.H.
STASTRIP.H	Definitions for STASTRIP.DLL. Identical to STASTRIP\STASTRIP.H.

Note in Table 2-2 that a number of the files in this directory are duplicates of those found in other directories. This is simply to provide you with an environment in which you can immediately compile any of the chapter-specific samples. To build the sample for any chapter, you must add this directory to those listed in your INCLUDE environment variable.

Libraries: The LIB Directory

Like the INC directory, the LIB directory is a repository for any .LIB that is useful to more than one sample. For the most part, the files found here are builds of their respective components, as listed in Table 2-3.

You can, of course, build each of these LIBs from the respective sources at your disposal, but you must first build BTTNCUR to build GIZMOBAR; you must build both of those and STASTRIP to build CLASSLIB; and you must build CLASSLIB to build most of the chapter samples. The builds provided on the companion disks are simply intended to save you the trouble of such interdependencies.

The BUILD Directory

As was mentioned earlier, the BUILD directory is the respository for builds of DLL and EXE samples. After installing the companion disks, this directory will contain the files shown in Table 2-4.

File	Purpose
BOOKUI.LIB	Import library for BOOKUI.DLL.
CLASSMDI.LIB	A Multiple Document Interface (MDI) build of the class libraries in CLASSLIB.
CLASSSDI.LIB	A Single Document Interface (SDI) build of the class libraries in CLASSLIB.

Table 2-3.
Contents of the LIB directory in the sample code.

File	Purpose
BOOKUI.DLL	A build of the OLE2UI library provided with the OLE 2 SDK specifically named for this book.
DATATRAN.DLL	A build of the Data Transfer object from the source code in \CHAP07\DATATRAN.
LNKASSIS.DLL	A build of the Link Assistant object from the source code in \CHAP12\LNKASSIS.

Table 2-4.
Contents of the BUILD directory in the sample code.

You should ensure that the BUILD directory is in your PATH command, because many samples depend on these files at run-time. Note that the versions of BTTNCUR.DLL and GIZMOBAR.DLL provided with the OLE 2 SDK are the same as the ones provided here, so you do not need to worry about the location of the BUILD directory in your path relative to the OLE 2 directory.

NOTE: Before you build any other projects described in this book, be sure to run the MAKEALL.BAT files in the \BTTNCUR, \GIZMOBAR, and \STASTRIP directories so that you have BTTNCUR.DLL, GIZMOBAR.DLL, and STASTRIP.DLL in the \BUILD directory and BTTNCUR.LIB, GIZMOBAR.LIB, and STASTRIP.LIB in the \LIB directory. These files must be built in this order because GizmoBar needs the Buttons & Cursors library. The MAKEALL.BAT file in the installation directory builds these files in the correct order for you.

Three Amigos: BttnCur, GizmoBar, and StatStrip

To fully demonstrate all the user interface affected by in-place activation and to add some spice to the samples, we need a few slick controls, such as a toolbar and a status line. The GizmoBar, whose source code is in the GIZMOBAR directory, is an implementation of a typical toolbar control that

builds on code provided in BttnCur, a DLL that draws up to six states (for example, up, down, disabled, and so on) of toolbar buttons from a single bitmap image. The GizmoBar uses BttnCur to draw its buttons, but it is also capable of containing any other standard Windows control. The GizmoBar is not able to hold arbitrary custom controls, however.

NOTE: The code for BttnCur is a version 1.1 refinement of BttnCur 1.0 that was included with *The Windows Interface: An Application Design Guide* from Microsoft Press. Version 1.1 has two major feature enhancements—support for different display resolutions and full color control, which allows the standard black/white/gray buttons to change with the system colors.

The StatStrip control provides a rudimentary message bar that generally is placed at the bottom of frame windows. The StatStrip is capable of managing a number of strings and displaying one of those strings on request. It also provides an almost painless way of tracking menu selections and displaying the appropriate message for each item. If you are interested in this mechanism, please study the sources in the STASTRIP directory.

NOTE: Sources for both BttnCur and GizmoBar are included with the OLE 2 SDK. The source code provided with this book is slightly and innocuously altered from the code in the OLE 2 SDK, but both sources build identical DLLs.

All three of these DLLs are implemented in straight C, mostly because they were projects that I wrote prior to writing this book. As I mentioned earlier, GizmoBar makes use of BttnCur, so you must build the latter to build the former.

Class Libraries: The CLASSLIB Directory

I mentioned earlier that I did not use a real C++ class library or an "application frameworks" to implement these samples. However, I still wanted to keep much of the mundane Windows code out of the way as we work through OLE 2, so I concentrated as much code as was reasonable into my own class library, which you'll find in the CLASSLIB directory. This code will compile into either an MDI version (CLASSMDI.LIB) or an SDI version (CLASSSDI.LIB), builds of which are provided in the LIB directory. The primary *include* file for these libraries is CLASSLIB.H, with resource definitions in CLASSRES.H.

Both Cosmo and Patron, as well as a few other samples in this book, make use of CLASSLIB. CLASSLIB essentially provides the framework for a simple application so that when we need to add a feature or a customization,

we only need to override the applicable virtual functions in the default CLASSLIB classes with a more specific implementation. To see how much CLASSLIB provides on its own, you only need to create a "frame window" object, initialize it, and tell it to start spinning in a message loop.

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

/*
 * WinMain
 *
 * Purpose:
 *  Main entry point of application. Should register the app class
 *  if a previous instance has not done so and do any other one-time
 *  initializations.
 */

int PASCAL WinMain (HINSTANCE hInst, HINSTANCE hPrev
    , LPSTR pszCmdLine, int nCmdList
    {
    LPCFrame        pFR;
    FRAMEINIT       fi;
    WPARAM          wRet;

    //Attempt to allocate and initialize the application
    pFR=new CFrame(hInst, hPrev, pszCmdLine, nCmdShow);

    fi.idsMin=IDS_STANDARDFRAMEMIN;
    fi.idsMax=IDS_STANDARDFRAMEMAX;
    fi.idsStatMin=IDS_STANDARDSTATMESSAGEMIN;
    fi.idsStatMax=IDS_STANDARDSTATMESSAGEMAX;
    fi.idStatMenuMin=ID_MENUFILE;
    fi.idStatMenuMax=ID_MENUHELP;
    fi.iPosWindowMenu=WINDOW_MENU;
    fi.cMenus=CMENUS;

    //If we can initialize pFR, start chugging messages
    if (pFR->FInit(&fi))
        wRet=pFR->MessageLoop();

    delete pFR;
    return wRet;
    }
```

You can find code identical to this in CHAP02\SKEL, which builds a skeletal application based on CLASSLIB, complete with toolbar and status line, using (of course) GIZMOBAR.DLL and STASTRIP.DLL.

With this class library, the code to implement both Cosmo and Patron deals almost exclusively with the special features of each application. In this way, we keep the typical windowing code out of our way to show only the application features and how OLE 2 affects them. Throughout this book, CLASSLIB will remain unaltered—all modifications to accommodate OLE 2 will be made only in the respective application's source code.

All of the C++ classes defined in CLASSLIB are shown in Table 2-5. Note, however, that this class library is not intended to be a basis for your own application (but, of course, there's nothing stopping you). I do not intend to revise this library, and I certainly will not be able to provide the level of product support for this code that you would get with a real class library from a reputable tools vendor. I do encourage you to use a professional development environment in your own endeavors to produce applications.

Class	Purpose
CStringTable	Loads a range of strings from the application's resources into memory and provides an overloaded [ ] (array lookup) operator to access those strings.
CWindow	Base class for other window-related classes.
CGizmoBar	A C++ wrapper class for the control implemented in GIZMOBAR.DLL.
CStatStrip	A C++ wrapper class for the control implemented in STASTRIP.DLL.
CFrame	Creates and manages a frame window that owns a menu, toolbar, and a client window. Compiles differently for MDI and SDI cases.
CClient	Creates and manages a client window identical to an MDI client window for MDI cases. Under SDI, provides a client window that responds to the MDI messages, isolating the rest of the application from many MDI/SDI differences.
CDocument	Creates and manages a document window inside the client window. The window is either an MDI child or a simple child window, depending on the build.

Table 2-5.
C++ classes in CLASSLIB and their uses.

CLASSLIB also contains resource files necessary for building a skeletal application. These resource files are the ones used by CHAP02\SKEL. These are not compiled into the library itself, but reside here as templates for applications using this library.

Note the use of one macro in CLASSLIB.H that might appear odd: *PSZ(i)* where *i* is an integer string identifier, always with an *IDS_* prefix. The *PSZ* macro simplifies the lookup of the string of that index in a *CStringTable* object that manages stringtable resources. When reading the code in this book, read *PSZ* as meaning "this string from the stringtable." A quick look in the .RC file for the relevant sample will show you exactly which string is being referenced.

Interface Templates: The INTERFAC Directory

C++ programmers: Don't get your hopes up. The INTERFAC directory contains a large number of .CPP and .H files—one for each interface—that we'll explore throughout this book. These are not official "C++ templates," rather they are interface templates that are simply source files containing a stubbed or default implementation for each interface. In some cases, a file will contain a complete implementation of a specific type of Windows Object, implemented using a C++ object class. In most other cases, the files simply contain stubbed functions that are meant to serve as a respository for source code from which you can copy and paste into your own applications, an approach that requires much less work than typing all the function headers themselves. You can easily customize this code with a few quick search-and-replace passes in your favorite editor.

Chapter Sources: The CHAP*xx* Directories

The specific source code related to a specific chapter is found in the CHAP*xx* directories, where *xx* ranges from 2 through 16. I will not show the code for any complete sample in any chapters because the code is too long for such a listing. At times I will show the entire contents of a specific file relevant to the discussion, but I will not show much in the way of make files, .DEF files, icons or bitmaps, resource scripts, and even some include files.

Most of the code in the CHAP*xx* directories are different revisions of Cosmo and Patron as they evolve throughout the book. The initial versions of both of these applications are provided in the CHAP02\COSMO and CHAP02\PATRON directories, with each application discussed in more detail later. As these applications evolve throughout this book, we'll modify many small parts of different source files (adding source files as well). These modifications are consistently marked with two comments: *//CHAPTERxxMOD* and *//End CHAPTERxxMOD*, where *xx* is the relevant chapter number. These comment delimiters will help you see which changes I had to make

to both header (.H) and source (.CPP) files in our pursuit of OLE 2 nirvana. What I primarily show in each chapter is the code around the blocks, set off with these comments. So, for example, if you want to see which variables I added to a class to support a specific feature, look in the .H file, and you'll see the new ones between these comments.

In addition, most CHAP*xx* directories contain a CHAP*xx*.REG file. These are plain text files containing chapter-relevant entries for the system Registration Database. Some files contain duplicate entries from previous chapters because some samples depend on samples and builds from previous chapters. This redundancy ensures that the proper entries exist if you skip a chapter.

So before attempting to run any sample in a chapter, you must merge the contents of the appropriate .REG file with the existing Registration Database by using the Windows 3.1 REGEDIT program. In some cases, the .REG file from a later chapter will replace some of the entries made with an earlier .REG file, which is why there is not just one master file for the entire book.

Keep in mind as you examine the code that I designed it in such a way that code changes or additions made to accommodate OLE 2 occur in one place. This is the same idea as centralizing drawing code in a window in its WM_PAINT message handling: Any other code that wants to draw something merely changes the state of the data and causes a repaint. This design, as well as other designs in the sample code, are my personal choices and are not meant to represent Truth. "If it's Truth you're interested in," as Dr. Indiana Jones would remind us, "Dr. Tyree's philosophy class is right down the hall."

Cosmo: A Graphical Editor (with Apologies to No One in Particular)

Cosmo is an application with a silly name that does nothing important. Despite its limited value, Cosmo is a typical application that creates some kind of graphical data—in this case, an image called a polyline. The polyline is simply any number of points between 0 and 20 connected by lines, as shown in Figure 2-3, which is managed by a C++ class in Cosmo called *CPolyline*.

The user is able to add up to 20 lines by clicking in the Polyline region—the Polyline adds the points to an array of 20 POINT structures and increments a point count. The user can reverse added points by using Undo, which simply decrements the number of points drawn and repaints. The user can also change both the line and background colors, as well as change the line style, but these operations are not reversible. All commands are available from either menus or the toolbar. Cosmo also sports a simple status line at the bottom of its window.

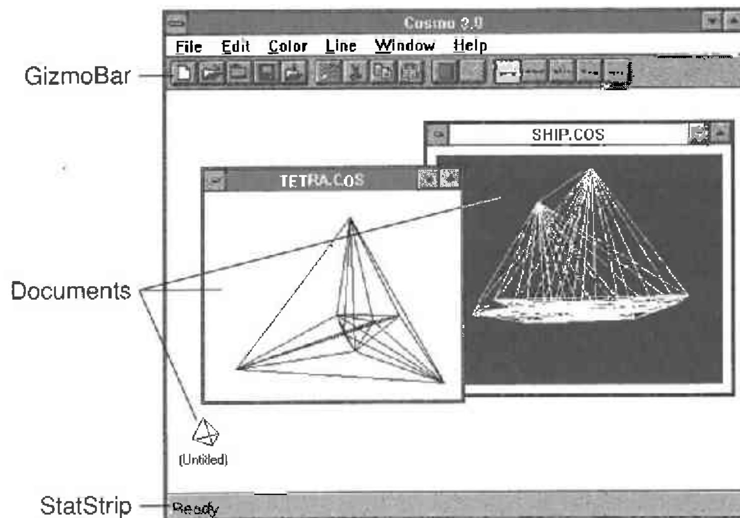


Figure 2-3.
Multiple-document version of Cosmo, with several open Polylines.

I provide two versions of Cosmo for use in illustrating object conversion and emulation between OLE 1 and OLE 2 servers, as discussed in Chapter 14. CHAP02\COSMO contains the source code for the C++ version build on CLASSLIB with a version 2 number. CHAP02\COSMO10 contains the source code for an earlier version written in straight C as an OLE 1 server. Anytime I refer to Cosmo in this book, I'm referring to version 2 unless I specifically state otherwise.

Cosmo performs traditional Windows file I/O, uses the Windows API to support the clipboard, and handles conversion to and from its version 1 file format. Both file formats use the .COS extension. A few sample Cosmo files can be found in CHAP02\COSFILES. In any case, Cosmo certainly lacks a few features (such as printing capability, Help, and maybe some real Undo functionality) that will keep it from being something I could sell. It does, however, maintain those elements that you would typically find in most applications of a higher caliber.

Cosmo will follow a course of evolution that will take it from a standard application for Windows to an OLE 2 application. Starting with Chapter 4, we'll also begin to separately develop a version of Cosmo's *CPolyline* object as an OLE 2 component object in a DLL. We'll also create a separate modified copy of Cosmo called Component Cosmo, or CoCosmo, that will use this component Polyline object so as to appear indistinguishable from the self-contained Cosmo. Using Polyline, we'll explore how various OLE 2 features can affect such a DLL, while at the same time illustrating those features in the

self-contained Cosmo .EXE. Both cases are important to illustrate, and both follow paths detailed in Table 2-6.

Chapter	Features
DLL Object Path	
4	Polyline is split from Cosmo into a component object DLL. A version of Cosmo called CoCosmo is created to demonstrate how to instantiate and use such an object.
5	Polyline begins to use compound files. CoCosmo is modified to follow the changes to Polyline.
6	Polyline implements a data object interface to provide uniform data transfer. CoCosmo is modified in Chapters 7 and 8 to use this data object interface to implement clipboard and drag-and-drop support.
11	Polyline is upgraded to a full compound document object that supports embedding. Polyline essentially becomes an embedded object server.
16	Polyline becomes capable of in-place activation.
Application Object Path	
5	Cosmo is converted to use compound files.
7	Cosmo implements a data object and converts clipboard transfers to using data objects.
8	Cosmo adds drag-and-drop functionality.
10	Cosmo is modified to support compound documents as an embedded object server.
11	An object handler DLL is created for Cosmo's embedded objects.
13	Cosmo is capable of providing compound document linked objects.
14	Cosmo becomes capable of converting and emulating OLE 1 objects from Cosmo version 1.
16	Cosmo becomes capable of in-place activation.

Table 2-6.
Evolution of the Cosmo application by chapter.

Polyline will compile into POLY*xx*.DLL, where *xx* is the appropriate chapter number of the build. Likewise, Cosmo will compile into COSMO*xx*.EXE. This naming scheme is intended to avoid naming conflicts when all the files are copied to the BUILD directory.

Patron: A Page Container (with Apologies to Merriam-Webster)

When I created the first version of this application for OLE 1, we called containers "clients," so a brief encounter with a thesaurus generated the name

Patron. In this case, *patron* is defined as either "one who uses the services of another establishment" or "the proprietor of an establishment (such as an inn)."[2] After all, as a container, Patron will use the implementations of compound document objects and provide a place (a document) in which they stay. Patron seems a better choice than another butchered version of "container," which doesn't fit an 8-character filename anyway. Essentially, Patron is a place to store various objects, such as bitmaps, metafiles, sounds, or spreadsheets—all of which Patron refers to as "tenants."

Patron's documents are pages that match the size and orientation of whatever printer setup you choose. You can add or delete pages and navigate through them, as well as scroll the view of the current page around in the document window. These commands are available from the menu or from a toolbar, as shown in Figure 2-4. Like Cosmo, Patron also sports a status line because we'll eventually make use of it in demonstrating in-place activation.

Aside from features for changing the number of pages or navigating through them, the only meaningful commands that the initial version of Patron (in CHAP02\PATRON) supports are Printer Setup and Print. Printer Setup lets you change size and orientation as you can with any real application. Print will actually pump out a printed page for every page you've created, complete with page number. Patron also draws a rectangle on the page so that you can see the printable boundaries. How exciting can it get?

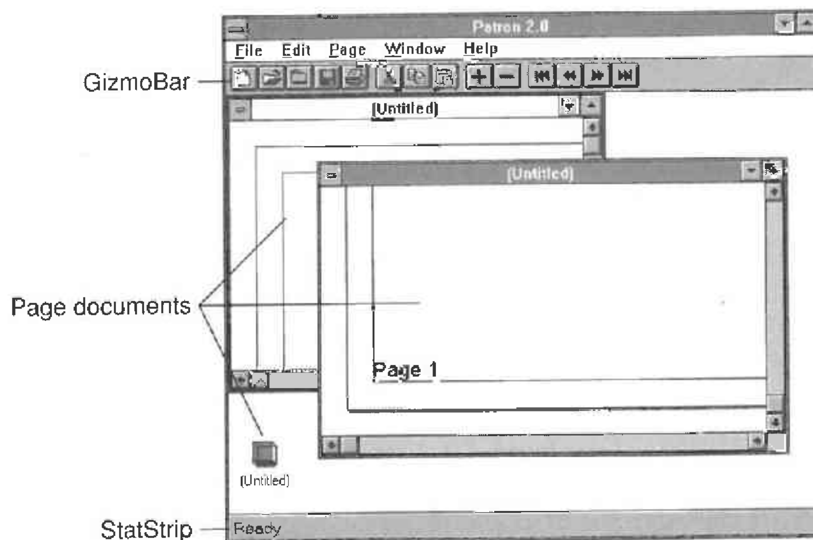


Figure 2-4.
Multiple-document version of Patron with several open documents.

2. *Webster's Ninth New Collegiate Dictionary*, Merriam-Webster, 1987.

Well, features stop there, as you might surmise from the code in CHAP02\PATRON. Patron's only purpose in life is to become an OLE 2 container application. Patron will be used to demonstrate how to write a relatively new application to take advantage of OLE 2 technologies. I did not bother to implement any file I/O for Patron because it will use compound files beginning in Chapter 5. I also didn't bother to make Patron capable of pasting metafiles and bitmaps from the clipboard because that would require a horrendous amount of code to draw those formats and to somehow serialize them to a file; you might call it laziness, but I call it planning.

Because we programmers instinctively try to avoid as much work as possible, we'll use functionality that OLE 2 already provides to add metafile and bitmap capabilities. As we'll see in the chapters ahead, OLE 2 already knows how to display and serialize these formats, so we need not consider writing such code ourselves. How convenient! We then add a little more code to enable Patron to contain compound document objects and work from there to in-place activation. As we progress through the chapters, we'll add various features to Patron, as shown in Table 2-7.

Chapter	Features
5	Patron adds file I/O using compound files.
7	Patron implements clipboard functions using a data object. It pastes metafiles and bitmaps using OLE 2 for drawing and serialization to a compound file.
8	Patron adds drag-and-drop functionality.
9	Patron is made into a simple compound document container for embedded objects only.
12	Patron becomes capable of containing linked objects.
13	Patron handles linking to embedded object stored in its own documents.
14	Patron handles object conversion and emulation.
15	Patron becomes capable of in-place activation.

Table 2-7.
Evolution of the Patron application by chapter.

Like Cosmo, Patron will compile into PATRON*xx*.EXE, where *xx* is the appropriate chapter number of the build. This naming scheme is intended to avoid naming conflicts when all the files are copied to the BUILD directory.

Building and Testing Environment

As I've mentioned before, the sample code in most of the directories depends on various files in the INC and LIB directories. Running the samples require some of the DLLs that are found in the LIB directory, as well. For these reasons, you need to make the following changes to your environment variables:

1. Add the INC directory to your INCLUDE path so that the compiler can locate the book's include files referenced with *#include <file>*.
2. Add the LIB directory to your LIB path so that the linker can find the libraries referenced in various make files.
3. Add the BUILD directory to your PATH so that when you run samples from the chapters they will be able to load the necessary DLLs.

In addition, note that the .REG files included with each chapter do not provide full pathnames to DLLs and EXEs referenced in those Registration Database entries, which is why you should add the BUILD directory to your PATH. Otherwise, you can modify the Registration Database to include full pathnames to each compiled DLL and EXE as needed. For debugging purposes, I recommend the latter approach. If you merely want to compile and run the samples quickly, I recommend the former approach.

There are two environment variables that affect compilations, as shown in Table 2-8. When the SDI variable is set to *1*, builds that are sensitive to that variable (Cosmo and Patron always are) will build into an SDI directory under the relevant source code directory (for example, CHAP02\COSMO\SDI). When the SDI variable is clear, builds will end up in the MDI directory under the relevant source tree. In addition, if you set the RETAIL variable to 1, you will build a nondebugging version in the appropriate SDI or MDI directory for whatever SDI or MDI option is set, wiping out the previous build in that same directory.

Variable	Purpose
SET SDI=1	Sets the SDI flag to build SDI versions.
SET SDI=	Clears the SDI flag to build MDI versions.
SET RETAIL=1	Builds nondebug versions using optimizations and eliminating debugging symbols.
SET RETAIL=	Builds debug versions with symbols and no optimization.

Table 2-8.
Build options controlled through environment variables.

In the sample code on your disk, you will find a number of files called MAKEALL.BAT. In any given directory, the file will completely rebuild all the samples visible in that directory. For example, the MAKEALL.BAT file in BTTNCUR will build BTTNCUR.DLL and a small demonstration program, BCDEMO.EXE. The MAKEALL.BAT in CHAP02\COSMO will build both MDI and SDI versions of Cosmo into CHAP02\COSMO\MDI and CHAP-02\COSMO\SDI. The MAKEALL.BAT in any CHAP*xx* directory will build all the samples—both SDI and MDI versions—for that chapter. For example, the one in CHAP02 will build MDI and SDI versions of Skel, Cosmo, and Patron, as well as the single SDI version of Cosmo version 1.

For your convenience, the MAKEALL files will redirect all error output from any compilation into a file called ERR in the same directory as the build DLL or EXE and will also concatenate all error output from all builds into BUILD\ERR. This provides a convenient record of any compilation problems. In addition, MAKEALL will copy the builds of all the DLLs to the BUILD directory along with MDI versions of all EXEs. SDI versions of EXEs are copied into BUILD\SDI.

Finally, the MAKEALL.BAT in the directory where you installed the sample code will rebuild every sample for the book, including all the DLLs and libraries in both MDI and SDI versions, for whatever debugging or retail version you have indicated through the RETAIL environment variable. It will also install builds to the INC, LIB, and BUILD directories as appropriate.

It's not a bad idea to install the samples at the end of your work day or before lunch and run MAKEALL before you leave your office. It will take some time to compile everything. Plenty of time to have a great lunch.

CHAPTER THREE

OBJECTS AND INTERFACES

object *n* **1** ***syn*** THING, article; ***rel*** doodad; gadget **2** ***syn*** THING, being, entity, individual, material, matter, stuff, substance.[1]

"Objects solve everything," or so you might have heard. If an object is a *thing*, how does one *thing* solve other *things*? The answer is it doesn't. Things don't solve, people solve. The belief that "object virtues" solve all your programming problems is what some friends of mine classify as "objects on the brain." They suggest that you attend meetings of your local OOPaholics Anonymous.

Using object-oriented languages to write applications and operating systems is only a matter of convenience if the ideas you want to express in that code are best done in such a language. But C++ programmers will tell you great stories about how C++ solved many problems they encountered in C but also introduced a whole new class of unique problems. For one thing, your language of choice has never simplified design—it has only made the implementation of many designs faster and more robust. I wrote the code in this book in C++ for such conveniences.

So just what is an object? No doubt everyone reading this book has a different idea about the term *object.* Objects are becoming so commonplace in just about every facet of computing that it has become difficult to understand what the word *object* means in a variety of contexts. Object models appear in places regardless of their relationships to any sort of object-oriented programming model. This chapter will attempt to clarify exactly what we mean by a Windows Object (note the capitalization to make the distinction) and what we mean by the interfaces that such an object supports. The standardized specifications of both are part of OLE 2's Component Object Model.

Windows Objects are slightly different from what C++ programmers might be used to. For instance, Windows objects do not allow direct access to data. Windows Objects can also be used and implemented in C or any other

1. *Webster's Collegiate Thesaurus*, Merriam-Webster, Inc., 1976.

language—that is, an object-oriented language is not necessary, only more convenient, to express object-oriented ideas.

We also have to distinguish between the object implementation and the object user, which this book will refer to as the "user" in programming contexts. The term *user* here should not be confused with the end user, a person who will see only the features you are implementing in your applications and will generally not be aware of your programming constructs.

This chapter will look at objects and interfaces in both C and C++ without delving deeply into OLE 2 itself. The first sets of code we'll see don't even use *#include* in any of the OLE 2 include files, but they still implement what we mean by a Windows Object—that is, something with interfaces. With a solid understanding of these fundamentals, we can move forward into seeing what is required for more complex Windows Objects with more useful capabilities. Note that much of the background, beginning with the section "*IUnknown*, the Root of All Evil," leads directly into Chapter 4.

I want to stress that an object as presented here is not a compound document (linked or embedded) object. We're not yet talking about specific applications such as containers. Much of the information from this point through Chapter 8 deals with topics completely outside the realm of Compound Document technology. So, as Yoda might suggest, "clear your mind of questions" and be prepared to learn what we mean by *object* in the cosmos of OLE 2.

Do objects solve everything? No. Do OLE 2 and its object model solve everything? No. OLE 2 intends to *simplify the expression* of object-oriented ideas under Windows. It does not intend to somehow make application or system design fall freely, like manna, from heaven. If it could, we would not have to worry about the national debt.

The Ultimate Question to Life, the Universe, and Objects (with Apologies to Douglas Adams)

I know a Windows Object exists that is capable of specific functions. How do I obtain a pointer to that object? This question is a central theme in this book: This chapter and those that follow are concerned with specific types of objects, how you get a pointer to one, and what you can do with that object once you have the pointer. Each chapter generally deals with different object types (and how you identify those objects), the interfaces they support, techniques to obtain their pointers (for whatever code uses the object), and the specific functions you can call through those pointers. So the answer to *our* question (which is not "42," as it was in Douglas Adams's books) varies

with each subtechnology in OLE 2. Realize as well that a compound document object is only one type of Windows Object and that server applications are not the only object implementors. The fact is that almost all OLE 2 applications, regardless of what technologies they use, are both object users and object implementors.

To fully understand obtaining and using a Windows Object, we must first go back to a few even more fundamental questions. What is an object? To answer that question we must ask: What is an object class? To some, an object class may seem some mighty spiritual force divinely manifested in your include files. In reality, a class (and objects) can be described in terms that anthropologist Marvin Harris would call "practical and mundane,"[2] for in one way or another, a programmer or compiler has to reduce the notion of a class into code.

A *class*, in mundane terms, is the *definition* of a data structure (members) and the functions that manipulate that structure (member functions). The concept can be expressed in any programming language; C++, Smalltalk, and other such languages have merely formalized the notion. For example, C++ classes generally live in include files, such as this one, shown in Chapter 2:

```
class __far CAppVars
    {
    public:
        HINSTANCE   m_hInst;
        HINSTANCE   m_hInstPrev;
        LPSTR       m_pszCmdLine;
        int         m_nCmdShow;
        HWND        m_hWnd;

    public:
        CAppVars(HINSTANCE, HINSTANCE, LPSTR, int);
        ~CAppVars(void);
        BOOL FInit(void);
    };
```

A class is only a definition and carries no implementation, although classes in some languages may define default implementations that are not realized until there is some instantiation of the data structure that contains a function table and the variables of the class. We call that instantiated structure an *object*. In C++, objects are manifested in memory, as shown in Figure 3-1 on the next page.

The object has two components in memory: a function table, containing pointers to each member function (sometimes known as a *method*) defined in

2. In *Cows, Pigs, Wars, and Witches*, by Marvin Harris, Vintage Books, 1974.

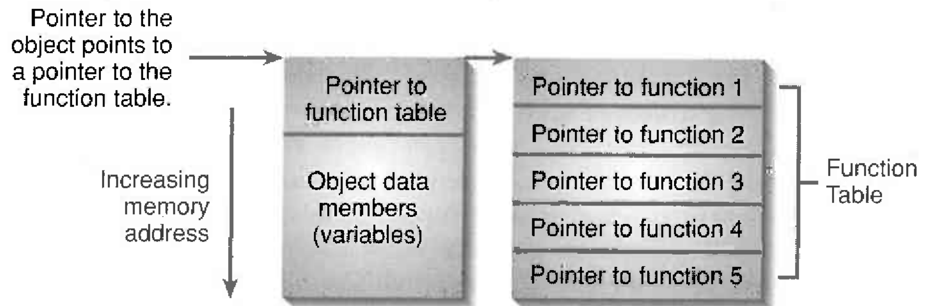


Figure 3-1.
A C++ object in memory is a data structure containing a pointer to the object's function table followed by the object's data. The function table is separate from the object structure itself, so a pointer to the object first points to a pointer to the function table.

the object's class, and a data block, containing the current values for each variable (or data member, sometimes known as a *property*). The user of the object generally has some reference[3] to this chunk of memory, which for the purposes of this book is always a pointer. The user obtains this reference by using some type of function call (direct or implied) in which that function allocates the block in memory, initializes the function table, and returns the reference to that memory to the user.

When the user has the reference to the block of memory, the user can call any of the functions in the object's function table and possibly access the object's variables, depending on the language being used. The single most important benefit is this: To call any of the functions defined in an object class, you must first have a reference to an instantiated object so that the functions have some data on which to operate. Without a reference to the object, you have no way to call one of the object's functions. Even with a pointer, the object can restrict your access to its variables or functions by means of language mechanisms such as *public* and *private* members in C++. In contrast, a non-object-oriented language such as C allows you to call any function with any garbage you want. Given a pointer to a data structure, there is nothing to keep you from partying all over those variables.

The OLE 2 notion of class is even more strict than the preceding general definition because the only accessible members of a class are specific groups of functions called *interfaces*. As mentioned in Chapter 1 and as shown in

3. *Reference* here does not necessarily mean a C++ reference.

Figure 3-2, an interface is a group of semantically related functions that are publicly accessible to the user of a Windows Object. An object's interface can really be viewed as only the function table part of an object in memory.

By themselves, interface definitions in OLE 2 are only virtual base classes, and thus they cannot be instantiated. In other words, they provide a convenient structure to lay over the top of a function table to provide more readable and maintainable names for each function.

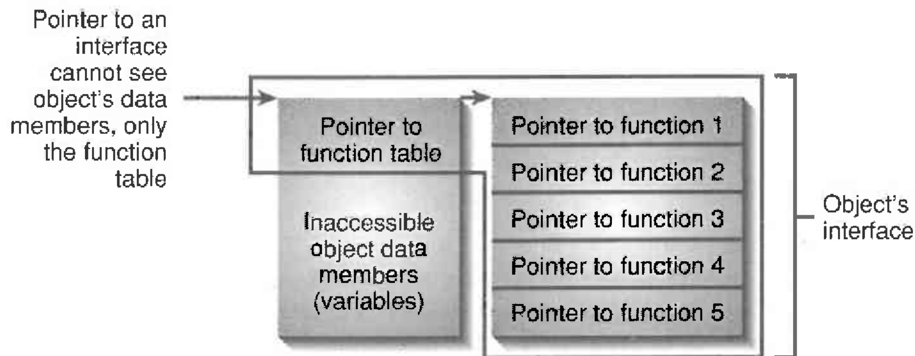


Figure 3-2.
A pointer to an interface can access only member functions in the object's function table.

An interface implementation, in pedestrian terms, is a block of memory containing an array of function pointers—that is, a function table. The interface definition itself simply provides names for each pointer in that table. When a user of a Windows Object obtains a pointer to an interface that an object supports, we say it has a pointer to an interface *on* that object. Again, that pointer does not provide access to the entire object; instead, it allows access to one interface on that object—that is, to one set of functions. Through an indirection on that pointer, the user calls a function of the object, as shown in Figure 3-3 on the next page.

As mentioned in Chapter 1, the user of a Windows Object has access to only one interface through one pointer, even when the object itself actually supports more than one interface—that is, implements more than one set of related functions and provides multiple function tables. Note that when we graphically represent an object with interfaces, we use a circle to represent each interface, as introduced in Chapter 1 and as shown in Figure 3-4 on the next page.

To use functions in a different interface on the same object, the user must obtain a second pointer to that other interface through the *QueryInterface* function, which is present in all interfaces. The section "*IUnknown*, the

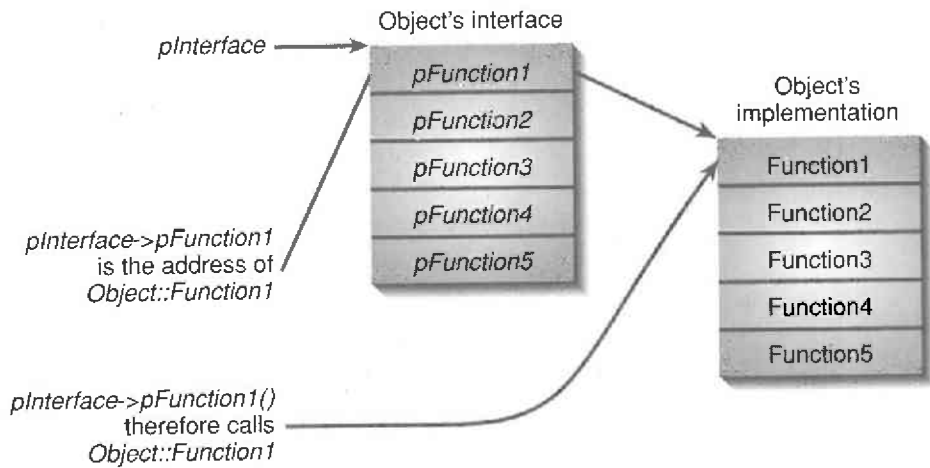


Figure 3-3.
Calling an interface member function. Note that the indirection through the pointer to the function table is not shown because C++ hides this extra step. The indirection is apparent, however, in C.

Root of All Evil'' later in this chapter explores *QueryInterface* in detail. *IUnknown* is a fundamental interface that all Windows Objects must support. (This is why in diagrams it's always placed above the object, as it is in Figure 3-4, instead of to the side, as other interfaces are.)

The function table itself is designed to have a layout that is identical to the one generated by many C++ compilers. Such a layout lets you use a single indirection (->) on the pointer to call an interface function. However, this does not force you to use C++ to program OLE 2; as I said in Chapter 2, C++ is

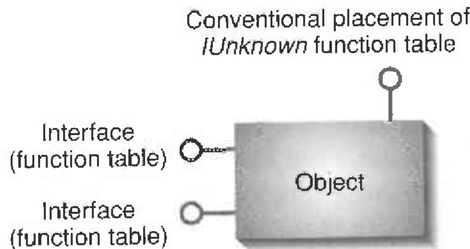


Figure 3-4.
Instead of always showing expanded function tables, interfaces are represented with a circle, or jack. By convention, IUnknown *is on top of the object, and all other interfaces are to the left or right.*

simply more *convenient.* Any object implementation is only required to provide separate function tables for each supported interface. How you choose to create each table will of course be different, depending on your language of choice, as the later section "A Simple Object in C and C++: *RECTEnumerator*" illustrates.

Because neither use nor implementation of a Windows Object is dependent on the programming language used, you can view OLE 2's object model, the Component Object Model, as a *binary standard.* This approach has a major advantage over other proposed object models. You can choose to implement in Visual Basic an object that is still usable from a C or C++ application as long as you can provide a pointer to your interface function tables. Microsoft has done us a wonderful service by not limiting our choice of programming tools or languages.

So back to the Ultimate Question: I know there's a Windows Object, but how do I obtain the first pointer to an interface on that object? The answer greatly depends on how you identify the Windows Object, but it can be reduced to four basic methods in OLE 2 for getting that pointer:

- Call an API function that creates an object of only one type—that is, the function will only ever return a pointer to one specific interface or object type.
- Call an API function that can create an object based on some class identifier and that returns any interface pointer you request.
- Call a member function of some interface that returns a specific interface pointer on another separate object.
- Implement interface functions on your own objects to which other object users pass their own interface pointers.

All these mechanisms are used by both OLE 2 applications and the OLE 2 libraries themselves. OLE 2 implements most of the API functions you'll use to obtain a pointer using the first two methods, but you might implement your own private API functions to accomplish similar ends. You will use the third method when you are the user of some object and have occasion to ask that object to create another object. You will use the fourth method for an object implementor whose user needs to provide the implementor with a pointer to the user's own objects. This last method is how two applications, such as a compound document container and a server, initiate a two-way dialog: Both applications implement specific (and different) objects and pass interface pointers to each other.

Windows Objects vs. C++ Objects

You might wonder why Windows Objects differ in many respects from C++ objects even though C++ is the most widely used object-oriented language for programming Windows. The overriding reason is that in C++ you might use only C++ objects that live and execute within your own application (EXE), possibly within DLLs (but at a price). On the other hand, you can use Windows Objects regardless of where they live and execute, be it in your own EXE, in a DLL (including the operating system itself), or in another EXE. In the future, Microsoft will enable Windows Objects to live and execute on another machine, a capability far out of reach of C++ objects.

Let's Go Traveling

Suppose I'm a C++ application that lives in Rugby, North Dakota (the geographic center of North America), and my application is bounded by the border of the continental United States, as illustrated in Figure 3-5. I can visit freely any of 48 states, no questions asked, by driving along an interstate. Access is fast and easy, although I am subject to the laws of each state I drive through. I can also drive into Canada or Mexico to buy their goods and use their services, but I do have to stop at the border and answer a few questions; travel is a little slower but still quite easy. In programming terms, I can freely use any object class within the boundaries of my application as long as I obey the access rights of those individual objects. I can also use objects implemented in DLLs, but there is more work involved in getting across the DLL boundary, even to my own DLL, such as Alaska.

I might live happily for a long time restricting my travels to a single continent. But there are six other continents and many other countries on the planet that I might want to visit. Getting there is not easy—I have to transfer flights, go through customs, and show my passport. If I want to travel to a distant destination, such as Antananarivo, Madagascar, I would have to fly to Chicago and then to London, switch carriers to get to Nairobi, Kenya, and then catch a final flight to Antananarivo. On each segment of my journey, I will probably fly on a different airline in a different airplane (or I might be forced to travel only by boat or train) and walk through customs offices in three different countries. If I step out of line anywhere, I might find myself in a prison on the other side of the globe.

As a C++ application, I would experience the same difficulty in using C++ objects implemented in other applications (countries) or code that is otherwise separated by a process boundary (oceans), as illustrated in Figure 3-6 on page 66. The best I can hope for is to become intimately familiar with

Figure 3-5.
Travel within North America is fairly painless.

the protocols and customs of each application along my way, knowledge that can apply only to those specific applications: When I want to use the services of a different application, I must learn another new interface. If time is not a luxury, I'll probably decide to visit only a few other countries.

OLE 2 offers you membership in the Windows Objects Club, which makes travel abroad much easier. The Windows Objects Club standardizes the protocol for visiting any other country, so you have to learn only one set of rules. The Windows Objects Club offers nonstop flights to many countries (Windows Objects in DLLs) and at worst one-stop flights to any other destination on the planet (Windows Objects in EXE applications). When you are a

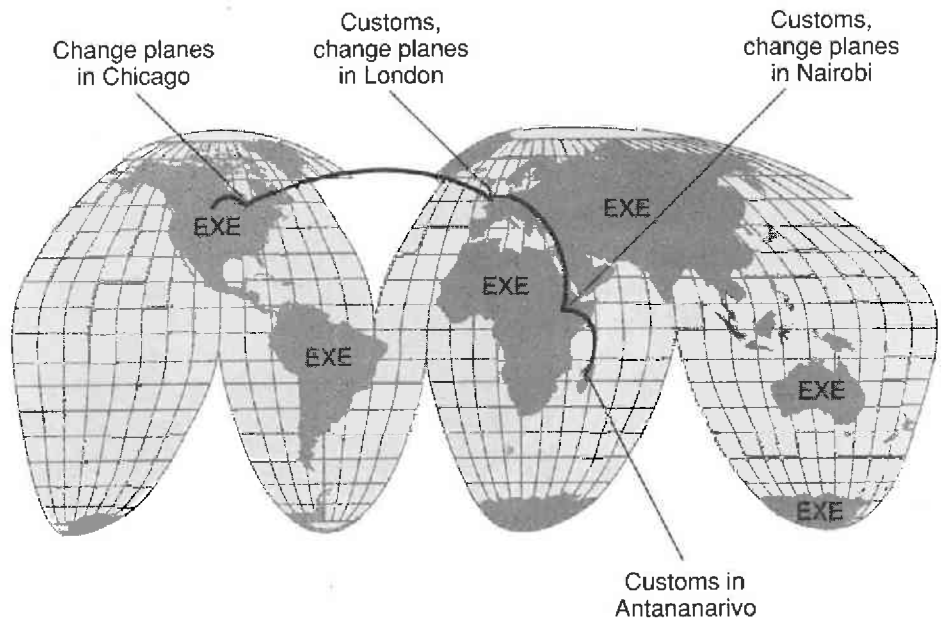


Figure 3-6.
Travel abroad involves much more time, effort, and knowledge.

member of the Windows Objects Club, travel is as easy as showing your membership card and hopping on a plane bound for whatever destination you choose. No matter where you are, the Windows Objects Club has a flight departing to any destination, as depicted in Figure 3-7.

In programming terms, you join the Windows Objects Club by using the various OLE 2 API functions to access specific objects without concern for where that object actually lives. Those API functions form the protocol you learn once; later, in Chapters 9 and beyond, we'll learn more about compound documents, which provide you with the benefit of a personal interpreter in any country you visit or with whom you do business. When we talk about in-place activation, we'll see how the Windows Objects Club can bring the country to you.

The Windows Objects Club today offers easy travel between all countries and continents on our little blue planet. In the future, this club will provide the same benefits to interplanetary and interstellar travel without even requiring you to reapply. More to the point, Windows Objects will become network aware and will allow you to use objects running on other machines, either on your local area network or even on a wide area network. Perhaps

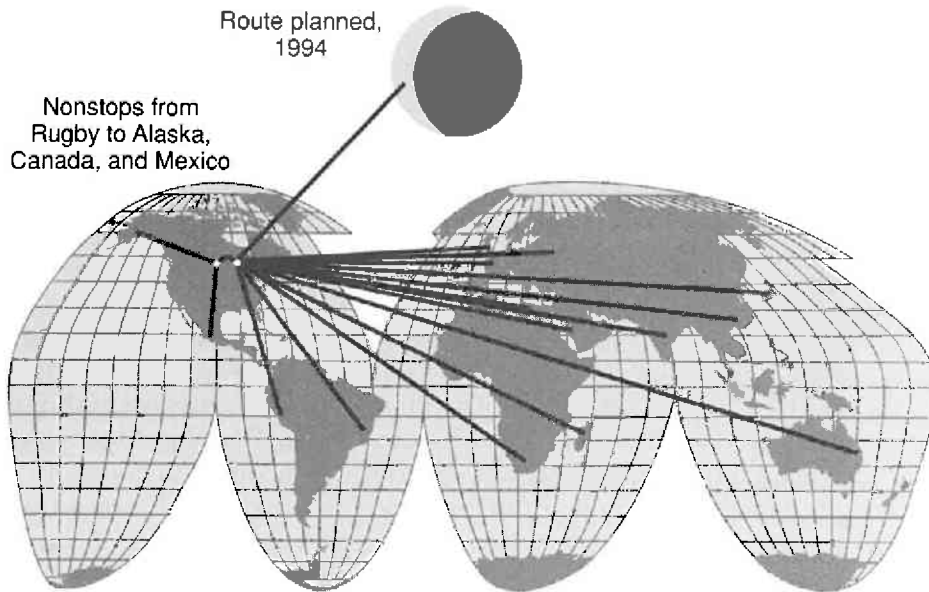


Figure 3-7.
The Windows Objects Club simplifies travel, and someday it will open more routes.

someday we'll have the PLAN (planetary network), letting you use objects that live on the moon, either figuratively or physically.

The purpose of this little exercise was to show that C++ objects are somewhat limited in scope because access to objects, being defined by the language, restricts you to objects that live in your own process space. Windows Objects, being defined by the system, open access to any object anywhere on your machine, and eventually on other machines as well.

Other Differences Between Windows Objects and C++ Objects

Because the location of an object's implementation varies so widely between C++ objects and Windows Objects, there are a number of other key implementation differences that affect programming:

- Class definition
- Object instantiation

- Object references
- Object destruction

Class Definition

C++ defines a class by using the *class* keyword, which generates a user-defined *type*. Members and member functions can be private, protected, or public. Furthermore, a C++ class can inherit from another class, thereby taking on all the characteristics (data and member functions) of that base class with the ability to override or expand select pieces of that base class.

A Windows Object is defined in terms of the interfaces it supports. All objects support at least one interface named *IUnknown*, which is discussed in the later section "*IUnknown*, The Root of All Evil"; support of this one interface qualifies the object as a Windows Object. The object user learns about other interfaces the object supports through member functions of *IUnknown*.

Windows Objects are all, therefore, at least of type *IUnknown* and can be treated as another type by means of a different interface. Because of this mechanism, there is no user-defined type associated with a Windows Object class as there is with a C++ class. In fact, there is no single way to identify a specific object. As we saw earlier, there are four general ways by which you can obtain a pointer to a Windows Object. Each technique has its own way of identifying the object. One of the techniques—identifying a Windows Object using a class identifier—is the closest analogy to a C++ method, but it is only one of the many ways to identify such objects.

Object Instantiation

C++ objects are instantiated by various means, such as declaring a variable of the object's type on the stack, declaring a global variable, or using the *new* operator on that type. Regardless of the actual technique used, C++ eventually calls the object's constructor.

Again, just as there are many ways to identify a Windows Object, there are many ways to instantiate an object. In some cases, you call a function to instantiate the object. In other cases, you don't directly instantiate an object, but you are given a pointer to one that something else already created. One of the most common techniques, described in Chapter 4, is to use a thing called a *class factory object* to instantiate a Windows Object, much as the *new* operator works for C++ objects. A class factory object represents a specific class identifier, is obtained by a specific OLE 2 function, and supports an interface named *IClassFactory*. The *IClassFactory* interface contains a member function named *CreateInstance*, to which you pass an identifier of the interface you want on that object. *IClassFactory::CreateInstance* is the logical equivalent of *new*.

Object References

C++ objects can be referenced through an object variable, an object reference (a special type in C++), or a pointer to the object. Because objects are always local (in your EXEs or DLLs), their instantiations can live anywhere in your process space. Through any variable, the user has access to any public members of the object or to private and protected members if the user and the object are friends.

As I hope I have beaten into your head by now, a Windows Object is *always* referenced through a pointer, not to the object itself but to an *interface.* This means that through a given interface pointer the user can access member functions only in that interface. The user can never have a pointer to the whole object (because there is no definition of *whole object*), so there is no access to data members and no concept of *friend.*

Through the *IUnknown* interface, a user can get at other interfaces that the object also supports, but that means obtaining a different pointer that refers to the same object. Each pointer to an interface points to a function table in the object, and each table contains only member functions for a specific interface, as shown in Figure 3-8 on the next page. Because every interface defined in OLE 2 is derived from *IUnknown,* it is not necessary to have an *IUnknown* pointer to query for other interfaces; you can use any other interface pointer as if it were an *IUnknown.*

When you have a pointer to an object's interface, you can call the interface's member functions just as you can call a member function of a C++ object through a pointer:

```
pObject->MemberFunction([parameters]);
```

Because a pointer to a Windows Object always points to a function table, such a pointer can also be used from C or from assembly code, not only from C++, as described in "A Simple Object in C and C++: *RECTEnumerator.*"

Object Destruction

In C++, you destroy an object created with *new* by calling the *delete* operator on an object pointer. Objects declared as stack variables are automatically freed by virtue of restoring the stack before returning from a function. In either case, the memory that the object occupied is freed, and the object's destructor function is called.

The function that frees a Windows Object and essentially calls its destructor is a member function called *Release.* This function is a member of the *IUnknown* interface, and so it is present in every interface you will ever

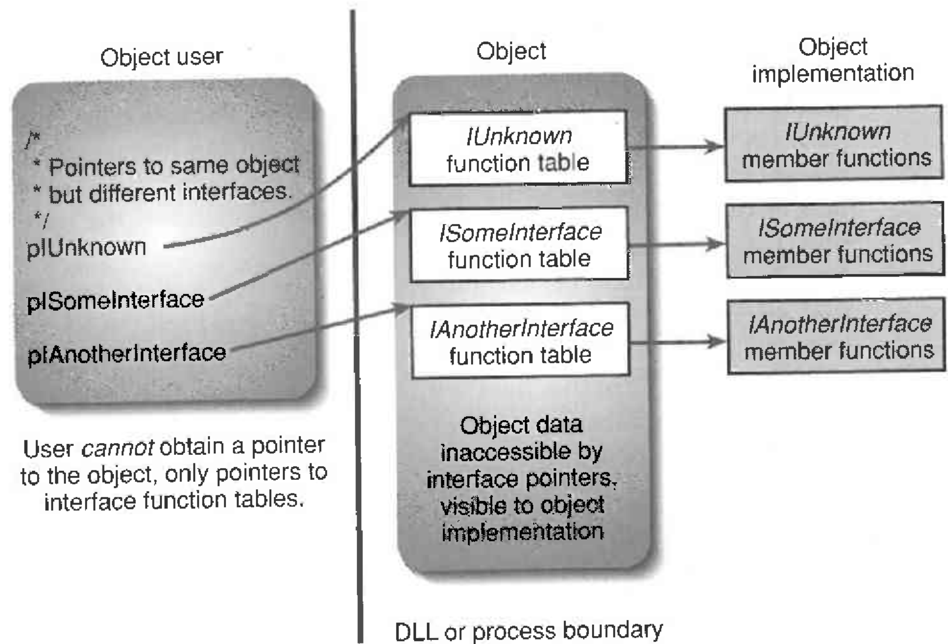


Figure 3-8.
Multiple interface pointers to an object reference unique function tables in the object but never reference the entire object itself.

obtain on any Windows Object. *Release*, however, is not as brutal as the *delete* operator, for as we'll see in the "Reference Counting" section later, the object might not actually be destroyed when *Release* is called. Internally, the object maintains a count of how many references exist to any of its interfaces. Creating an interface pointer increments the reference count, whereas *Release* decrements it. When the count is reduced to zero, the object frees itself, calling its own destructor.

A Simple Object in C and C++: *RECTEnumerator*

Windows Objects really can be written in any language; the most common are C and C++. As with any programming task, you need to choose a language that is well suited to the problems at hand, and C++ is the best suited for

expressing the ideas in OLE 2. Therefore, C++ is more natural and definitely more convenient for use in programming with Windows Objects. With a little more overhead, however, you can program just as effectively using C. The differences lie in how you create the function table for the object's interfaces and how you call the functions in those function tables. To illustrate the differences between the two languages, let's implement a type of object called an *enumerator.*

Enumerators are specific objects defined in OLE 2 that are used to communicate lists of information between another object and the user of that object, even when they are in different processes. For example, let's say you're using an object that represents a source of data—call it a data object—and you ask that object what data formats it supports (as we'll see in Chapter 6). The data object would create another independent enumerator that allows the user to iterate through the list of formats supported by the data object. The user of the data object also becomes a user of the enumerator, albeit through a different interface pointer.

An enumerator supports one of a set of interfaces prefixed with *IEnum.* Because the elements of the enumerator's list vary by context, OLE 2 defines a number of *IEnum<type>* interfaces where *<type>* is the name of the specific data structure used for each element in the list. OLE 2 also provides marshaling support for each standard *IEnum* interface. Each *IEnum* interface supports all member functions of *IUnknown* (of course) as well as four additional members to facilitate iteration over the list of elements:

Member	Result
Next	Returns the next *n* elements of the list starting at the current index.
Skip	Skips past *n* elements in the list.
Reset	Sets the current index to zero.
Clone	Returns a new enumerator object with the same state.

For this exercise, let's define a custom interface named *IEnumRECT* with all the functions in the preceding tale except *Clone.* Let's also define an object named *RECTEnumerator,* which implements that interface. The interface is defined in IENUM0.H, as shown in Listing 3-1 on page 73, which you'll find in the INC directory in the sample code. This file compiles differently for C and C++, depending on the *__cplusplus* symbol, which is defined only

when you are compiling for C++.[4] The C++ implementation of the *RECTEnumerator* object, in a program called ENUMCAP, is shown in Listing 3-2 on page 74, and the C implementation, in ENUMC, is shown in Listing 3-3 on page 83. Both samples are in the CHAP03 directory. Do not confuse the name *RECTEnumerator* for this object with the name of anything you might use to implement the object: It's merely a label.

Do ENUMC and ENUMCPP Do Anything?

When you run either ENUMC or ENUMCPP, you'll see no visible output no matter which menu commands you choose. You are *not* going crazy. Both programs are so intentionally boring on the outside that you should really want to run them in a debugger, which is where you can step through the code to see what is actually happening. These samples are intended for use in a debugger to illustrate interfaces in C and C++.

RECTEnumerator and the *IEnumRECT* Interface

The *RECTEnumerator* object supports one interface, named *IEnumRECT*, as shown in Listing 3-1, with the following member functions:

Member Function	Result
AddRef	Increments the reference count on the enumerator object.
Release	Decrements the reference count and frees the enumerator object when the reference count is zero.
Next	Returns the next *n* RECT structures starting at the current index.
Skip	Skips past *n* RECTs in the list.
Reset	Sets the current index to zero.

Because we have not yet examined *IUnknown* and because we want to keep this example as simple as possible, *IEnumRECT* borrows the two *IUnknown* members *AddRef* and *Release* but does not include *QueryInterface*. For the same reasons, we also eliminate the *Clone* member, which is part of standard *IEnum* interfaces.

4. The major C++ compilers, at least, define the __*cplusplus* symbol.

IENUM0.H

```
/*
 * Definition of an IEnumRECT interface as an example of the
 * interface notion introduced in OLE 2 with the Component Object
 * Model as well as the idea of enumerators.  This include file
 * defines the interface differently for C or C++.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _IENUM0_H_
#define _IENUM0_H_

//C++ Definition of an interface.
#ifdef __cplusplus

//This is the interface, a struct of pure virtual functions.
struct __far IEnumRECT
    {
    virtual DWORD AddRef(void)=0;
    virtual DWORD Release(void)=0;

    virtual BOOL  Next(DWORD, LPRECT, LPDWORD)=0;
    virtual BOOL  Skip(DWORD)=0;
    virtual void  Reset(void)=0;
    };

typedef IEnumRECT FAR *LPENUMRECT;

#else   //!__cplusplus

/*
 * A C interface is explicitly a structure containing a
 * long pointer to a virtual function table that we have to
 * initialize explicitly.
 */

typedef struct
    {
    struct IEnumRECTVtbl FAR *lpVtbl;
    } IEnumRECT;

typedef IEnumRECT FAR *LPENUMRECT;
```

Listing 3-1. *(continued)*

The IENUM0.H include file found in the shared INC directory.

Listing 3-1. *continued*

```
//This is simply a convenient naming
typedef struct IEnumRECTVtbl IEnumRECTVtbl;

struct IEnumRECTVtbl
    {
    DWORD (* AddRef)(LPENUMRECT);
    DWORD (* Release)(LPENUMRECT);
    BOOL  (* Next)(LPENUMRECT, DWORD, LPRECT, LPDWORD);
    BOOL  (* Skip)(LPENUMRECT, DWORD);
    void  (* Reset)(LPENUMRECT);
    };

#endif  //!__cplusplus

#endif //_IENUM0_H_
```

NOTE: The following files (ENUMCPP.H, ENUMCPP.CPP, IENUM.CPP) are for the C++ implementation of the Enum program.

ENUMCPP.H

```
/*
 * Enumerator in C++ Chapter 3
 * Definitions, classes, and prototypes for enumerator interface
 * example implemented in C++.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _ENUMCPP_H_
#define _ENUMCPP_H_

#include <ienum0.h>      //Found in shared include directory.
#include <book1632.h>

//Menu resource ID and commands
#define IDR_MENU                        1

#define IDM_ENUMCREATE                  100
#define IDM_ENUMRELEASE                 101
#define IDM_ENUMRUNTHROUGH              102
```

Listing 3-2.
The ENUM program implemented in C++.

(continued)

Listing 3-2. *continued*

```
#define IDM_ENUMEVERYTHIRD          103
#define IDM_ENUMRESET               104
#define IDM_ENUMEXIT                105

//ENUMCPP.CPP
LRESULT FAR PASCAL EXPORT EnumWndProc(HWND, UINT, WPARAM, LPARAM);

class __far CAppVars
    {
    friend LRESULT FAR PASCAL EXPORT EnumWndProc(HWND, UINT
        , WPARAM, LPARAM);

    protected:
        HINSTANCE       m_hInst;            //WinMain parameters
        HINSTANCE       m_hInstPrev;
        UINT            m_nCmdShow;

        HWND            m_hWnd;             //Main window handle
        LPENUMRECT      m_pIEnumRect;       //Enumerator interface

    public:
        CAppVars(HINSTANCE, HINSTANCE, UINT);
        ~CAppVars(void);
        BOOL FInit(void);
    };

typedef CAppVars FAR *LPAPPVARS;

#define CBWNDEXTRA          sizeof(LONG)
#define ENUMWL_STRUCTURE    0

//IENUM.CPP

//Number of rectangles that objects with IEnumRECT support (demo)
#define CRECTS      15

/*
 * A class definition, not provided by OLE, then inherits from
 * whatever interfaces it supports.  Multiple inheritance works
 * in this scenario as does the single inheritance shown here.
 */
class __far CImpIEnumRECT : public IEnumRECT
    {
    private:
```

(continued)

Listing 3-2. *continued*

```
        DWORD    m_cRef;         //Reference count
        DWORD    m_iCur;         //Current Enum position
        RECT     m_rgrc[CRECTS]; //RECTs we enumerate

    public:
        CImpIEnumRECT(void);
        ~CImpIEnumRECT(void);

        virtual DWORD AddRef(void);
        virtual DWORD Release(void);
        virtual BOOL  Next(DWORD, LPRECT, LPDWORD);
        virtual BOOL  Skip(DWORD);
        virtual void  Reset(void);
    };

typedef CImpIEnumRECT FAR *LPIMPIENUMRECT;

//Function that creates one of these objects
BOOL CreateRECTEnumerator(LPENUMRECT FAR *);

#endif //_ENUMCPP_H_
```

ENUMCPP.CPP

```
/*
 * Enumerator interface in C++ Chapter 3
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include <windows.h>
#include "enumcpp.h"

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    MSG         msg;
    LPAPPVARS   pAV;

    //Create and initialize the application.
    pAV=new CAppVars(hInst, hInstPrev, nCmdShow);
```

(continued)

Listing 3-2. *continued*

```
    if (NULL==pAV)
        return -1;

    if (pAV->FInit())
        {
        while (GetMessage(&msg, NULL, 0,0))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    delete pAV;
    return msg.wParam;
    }

LRESULT FAR PASCAL EXPORT EnumWndProc(HWND hWnd, UINT iMsg
    , WPARAM wParam, LPARAM lParam)
    {
    LPAPPVARS    pAV;
    RECT         rc;
    DWORD        cRect;

    COMMANDPARAMS(wID, wCode, hWndMsg);
    pAV=(LPAPPVARS)GetWindowLong(hWnd, ENUMWL_STRUCTURE);

    switch (iMsg)
        {
        case WM_NCCREATE:
            pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)
                ->lpCreateParams);

            SetWindowLong(hWnd, ENUMWL_STRUCTURE, (LONG)pAV);
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        case WM_COMMAND:
            switch (wID)
                {
```

(continued)

Listing 3-2. *continued*

```
            case IDM_ENUMCREATE:
                if (NULL!=pAV->m_pIEnumRect)
                    pAV->m_pIEnumRect->Release();

                CreateRECTEnumerator(&pAV->m_pIEnumRect);
                break;

            case IDM_ENUMRELEASE:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                if (0==pAV->m_pIEnumRect->Release())
                    pAV->m_pIEnumRect=NULL;

                break;

            case IDM_ENUMRUNTHROUGH:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                while (pAV->m_pIEnumRect->Next(1, &rc, &cRect))
                    ;

                break;

            case IDM_ENUMEVERYTHIRD:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                while (pAV->m_pIEnumRect->Next(1, &rc, &cRect))
                    {
                    if (!pAV->m_pIEnumRect->Skip(2))
                        break;
                    }

                break;

            case IDM_ENUMRESET:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                pAV->m_pIEnumRect->Reset();
                break;
```

(continued)

Listing 3-2. *continued*

```
                case IDM_ENUMEXIT:
                    PostMessage(hWnd, WM_CLOSE, 0, 0L);
                    break;
                }
            break;

        default:
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
        }

    return 0L;
    }

CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev
    , UINT nCmdShow)
    {
    //Initialize WinMain parameter holders.
    m_hInst     = hInst;
    m_hInstPrev = hInstPrev;
    m_nCmdShow  = nCmdShow;

    m_hWnd=NULL;
    m_pIEnumRect=NULL;

    return;
    }

CAppVars::~CAppVars(void)
    {
    //Free the enumerator object if we have one.
    if (NULL!=m_pIEnumRect)
        m_pIEnumRect->Release();

    return;
    }

BOOL CAppVars::FInit(void)
    {

    WNDCLASS    wc;

    if (!m_hInstPrev)
        {
```

(continued)

Listing 3-2. *continued*

```
        wc.style           = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc     = EnumWndProc;
        wc.cbClsExtra      = 0;
        wc.cbWndExtra      = CBWNDEXTRA;
        wc.hInstance       = m_hInst;
        wc.hIcon           = LoadIcon(m_hInst, "Icon");
        wc.hCursor         = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground   = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName    = MAKEINTRESOURCE(IDR_MENU);
        wc.lpszClassName   = "ENUMCPP";

        if (!RegisterClass(&wc))
            return FALSE;
        }

        m_hWnd=CreateWindow("ENUMCPP", "Enumerator in C++"
            , WS_MINIMIZEBOX | WS_OVERLAPPEDWINDOW
            , 35, 35, 350, 250, NULL, NULL, m_hInst, this);

    if (NULL==m_hWnd)
        return FALSE;

    ShowWindow(m_hWnd, m_nCmdShow);
    UpdateWindow(m_hWnd);

    return TRUE;
    }
```

IENUM.CPP

```
/*
 * Enumerator in C++ Chapter 3
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include <windows.h>
#include "ENUMCPP.H"

/*
 * CreateRECTEnumerator
 *
```

(continued)

Listing 3-2. *continued*

```
 * Purpose:
 *  Given an array of rectangles, creates an enumerator interface
 *  on top of that array.
 *
 * Parameters:
 *  ppEnum          LPENUMRECT FAR * in which to return the
 *                  interface pointer on the created object.
 *
 * Return value:
 *  BOOL            TRUE if successful, FALSE otherwise.
 */

BOOL CreateRECTEnumerator(LPENUMRECT FAR *ppEnum)
    {
    if (NULL==ppEnum)
        return FALSE;

    //Create the object storing a pointer to the interface
    *ppEnum=(LPENUMRECT)new CImpIEnumRECT();

    if (NULL==*ppEnum)
        return FALSE;

    //If creation worked, AddRef the interface
    if (NULL!=*ppEnum)
        (*ppEnum)->AddRef();

    return (NULL!=*ppEnum);
    }

CImpIEnumRECT::CImpIEnumRECT(void)
    {
    UINT        i;

    //Initialize the array of rectangles
    for (i=0; i < CRECTS; i++)
        SetRect(&m_rgrc[i], i, i*2, i*3, i*4);

    //Ref counts always start as zero
    m_cRef=0;

    //Current pointer is the first element.
    m_iCur=0;
```

(continued)

Listing 3-2. *continued*

```
    return;
    }

CImpIEnumRECT::~CImpIEnumRECT(void)
    {
    return;
    }

DWORD CImpIEnumRECT::AddRef(void)
    {
    return ++m_cRef;
    }

DWORD CImpIEnumRECT::Release(void)
    {
    DWORD       cRefT;

    cRefT=--m_cRef;

    if (0==m_cRef)
        delete this;

    return cRefT;
    }

BOOL CImpIEnumRECT::Next(DWORD cRect, LPRECT prc, LPDWORD pdwRects)
    {
    DWORD       cRectReturn=0L;

    if (NULL==pdwRects)
        return FALSE;

    *pdwRects=0L;

    if (NULL==prc || (m_iCur >= CRECTS))
        return FALSE;

    while (m_iCur < CRECTS && cRect > 0)
        {
        *prc++=m_rgrc[m_iCur++];
        cRectReturn++;
        cRect--;
        }
```

(continued)

Listing 3-2. *continued*

```
    *pdwRects=(cRectReturn-cRect);
    return TRUE;
    }

BOOL CImpIEnumRECT::Skip(DWORD cSkip)
    {
    if ((m_iCur+cSkip) >= CRECTS)
        return FALSE;

    m_iCur+=cSkip;
    return TRUE;
    }

void CImpIEnumRECT::Reset(void)
    {
    m_iCur=0;
    return;
    }
```

NOTE: The following files (ENUMC.H, ENUMC.C, and IENUM.C) are for the C implementation of the Enum program.

ENUMC.H

```
/*
 * Enumerator in C, Chapter 3
 *
 * Definitions, structures, and prototypes.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _ENUMC_H_
#define _ENUMC_H_

#include <ienum0.h>      //Found in shared include directory
#include <book 1632.h>

//Menu resource ID and commands
#define IDR_MENU                    1
```

Listing 3-3. (continued)
The ENUM program implemented in C.

Listing 3-3. *continued*

```
#define IDM_ENUMCREATE              100
#define IDM_ENUMRELEASE             101
#define IDM_ENUMRUNTHROUGH          102
#define IDM_ENUMEVERYTHIRD          103
#define IDM_ENUMRESET               104
#define IDM_ENUMEXIT                105

//ENUMC.C
LRESULT FAR PASCAL EXPORT EnumWndProc(HWND, UINT, WPARAM, LPARAM);

typedef struct tagAPPVARS
    {
    HINSTANCE       m_hInst;            //WinMain parameters
    HINSTANCE       m_hInstPrev;
    UINT            m_nCmdShow;

    HWND            m_hWnd;             //Main window handle
    LPENUMRECT      m_pIEnumRect;       //Enumerator interface
    } APPVARS, FAR *LPAPPVARS;

LPAPPVARS AppVarsConstructor(HINSTANCE, HINSTANCE, UINT);
void      AppVarsDestructor(LPAPPVARS);
BOOL      AppVarsFInit(LPAPPVARS);

#define CBWNDEXTRA          sizeof(LONG)
#define ENUMWL_STRUCTURE    0

//Number of rectangles that IEnumRECT objects support (for demo)
#define CRECTS      15

/*
 * In C we make a class by reusing the elements of IEnumRECT,
 * thereby inheriting from it, albeit manually.
 */

typedef struct tagIMPIENUMRECT
    {
    IEnumRECTVtbl FAR * lpVtbl;
    DWORD           m_cRef;         //Reference count
    DWORD           m_iCur;         //Current position
    RECT            m_rgrc[CRECTS]; //RECTs we enumerate
    } IMPIENUMRECT, FAR *LPIMPIENUMRECT;

/*
```

(continued)

Listing 3-3. *continued*

```
 * In C, you must separately declare member functions
 * with globally unique names, so prefixing with the class name
 * should remove any conflicts.
 */

LPIMPIENUMRECT  IMPIEnumRect_Constructor(void);
void            IMPIEnumRect_Destructor(LPIMPIENUMRECT);

DWORD           IMPIEnumRect_AddRef(LPENUMRECT);
DWORD           IMPIEnumRect_Release(LPENUMRECT);
BOOL            IMPIEnumRect_Next(LPENUMRECT, DWORD, LPRECT
                    , LPDWORD);
BOOL            IMPIEnumRect_Skip(LPENUMRECT, DWORD);
void            IMPIEnumRect_Reset(LPENUMRECT);

//Function that creates one of these objects
BOOL CreateRECTEnumerator(LPENUMRECT FAR *);

#endif //_ENUMC_H_
```

ENUMC.C

```
/*
 * Enumerator in C Chapter 3
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include <windows.h>
#include <malloc.h>
#include "enumc.h"

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    MSG         msg;
    LPAPPVARS   pAV;

    pAV=AppVarsConstructor(hInst, hInstPrev, nCmdShow);

    if (NULL==pAV)
        return -1;
```

(continued)

Listing 3-3. *continued*

```
    if (AppVarsFInit(pAV))
        {
        while (GetMessage(&msg, NULL, 0,0 ))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    AppVarsDestructor(pAV);
    return msg.wParam;
    }

LRESULT FAR PASCAL EXPORT EnumWndProc(HWND hWnd, UINT iMsg,
    WPARAM wParam, LPARAM lParam)
    {
    LPAPPVARS       pAV;
    RECT            rc;
    DWORD           cRect;

    COMMANDPARAMS(wID, wCode, hWndMsg);

    pAV=(LPAPPVARS)GetWindowLong(hWnd, ENUMWL_STRUCTURE);

    switch (iMsg)
        {
        case WM_NCCREATE:
            pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)
                ->lpCreateParams);

            SetWindowLong(hWnd, ENUMWL_STRUCTURE, (LONG)pAV);
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        case WM_COMMAND:
            switch (wID)
                {
                case IDM_ENUMCREATE:
                    if (NULL!=pAV->m_pIEnumRect)
                        {
```

(continued)

Listing 3-3. *continued*

```
                pAV->m_pIEnumRect->lpVtbl->Release(pAVM
                    ->m_pIEnumRect);
                }

            CreateRECTEnumerator(&pAV->m_pIEnumRect);
            break;

        case IDM_ENUMRELEASE:
            if (NULL==pAV->m_pIEnumRect)
                break;

            if (0==pAV->m_pIEnumRect->lpVtbl->Release(pAV
                ->m_pIEnumRect))
                pAV->m_pIEnumRect=NULL;

            break;

        case IDM_ENUMRUNTHROUGH:
            if (NULL==pAV->m_pIEnumRect)
                break;

            while (pAV->m_pIEnumRect->lpVtbl->Next(pAV
                ->m_pIEnumRect, 1, &rc, &cRect))
                ;

            break;

        case IDM_ENUMEVERYTHIRD:
            if (NULL==pAV->m_pIEnumRect)
                break;

            while (pAV->m_pIEnumRect->lpVtbl->Next(pAV
                ->m_pIEnumRect, 1, &rc, &cRect))
                {
                if (!pAV->m_pIEnumRect->lpVtbl->Skip(pAV
                    ->m_pIEnumRect, 2))
                    break;
                }
            break;

        case IDM_ENUMRESET:
            if (NULL==pAV->m_pIEnumRect)
                break;

            pAV->m_pIEnumRect->lpVtbl->Reset(pAV
                ->m_pIEnumRect);
            break;
```

(continued)

Listing 3-3. *continued*

```
                case IDM_ENUMEXIT:
                    PostMessage(hWnd, WM_CLOSE, 0, 0L);
                    break;
                }
            break;

        default:
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
        }

    return 0L;
    }

LPAPPVARS AppVarsConstructor(HINSTANCE hInst, HINSTANCE hInstPrev
    , UINT nCmdShow)
    {
    LPAPPVARS       pAV;

    pAV=(LPAPPVARS)_fmalloc(sizeof(APPVARS));

    if (NULL==pAV)
        return NULL;

    pAV->m_hInst     =hInst;
    pAV->m_hInstPrev =hInstPrev;
    pAV->m_nCmdShow  =nCmdShow;

    pAV->m_hWnd=NULL;
    pAV->m_pIEnumRect=NULL;

    return pAV;
    }

void AppVarsDestructor(LPAPPVARS pAV)
    {
    //Free any object we still hold on to
    if (NULL!=pAV->m_pIEnumRect)
        pAV->m_pIEnumRect->lpVtbl->Release(pAV->m_pIEnumRect);

    if (IsWindow(pAV->m_hWnd))
        DestroyWindow(pAV->m_hWnd);

    _ffree((LPVOID)pAV);
```

(continued)

Listing 3-3. *continued*

```
    return;
    }

BOOL AppVarsFInit(LPAPPVARS pAV)
    {
    WNDCLASS    wc;

    if (!pAV->m_hInstPrev)
        {

        wc.style          = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc    = (WNDPROC)EnumWndProc;
        wc.cbClsExtra     = 0;
        wc.cbWndExtra     = CBWNDEXTRA;
        wc.hInstance      = pAV->m_hInst;
        wc.hIcon          = LoadIcon(pAV->m_hInst, "Icon");
        wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
        wc.lpszClassName  = "ENUMC";

        if (!RegisterClass(&wc))
            return FALSE;
        }

        pAV->m_hWnd=CreateWindow("ENUMC", "Enumerator in C"
            , WS_MINIMIZEBOX | WS_OVERLAPPEDWINDOW
            , 35, 35, 350, 250, NULL, NULL, pAV->m_hInst, pAV);

    if (NULL==pAV->m_hWnd)
        return FALSE;

    ShowWindow(pAV->m_hWnd, pAV->m_nCmdShow);
    UpdateWindow(pAV->m_hWnd);

    return TRUE;
    }
```

IENUM.C

```
/*
 * Enumerator in C, Chapter 3
 * Implements the IMPIENUMRECT structure and functions (an object).
 */
```

(continued)

Listing 3-3. *continued*

```
#include <windows.h>
#include <malloc.h>
#include "enumc.h"

//We have to explicitly define function table for IEnumRECT in C
static IEnumRECTVtbl  vtEnumRect;
static BOOL           fVtblInitialized=FALSE;

/*
 * CreateRECTEnumerator
 *
 * Purpose:
 *  Given an array of rectangles, creates an enumerator interface
 *  on top of that array.
 *
 * Parameters:
 *  ppEnum          LPENUMRECT FAR * in which to return the interface
 *                  pointer on the created object.
 *
 * Return value:
 *  BOOL            TRUE if successful, FALSE otherwise.
 */

BOOL CreateRECTEnumerator(LPENUMRECT FAR *ppEnum)
    {
    if (NULL==ppEnum)
        return FALSE;

    //Create the object storing a pointer to the interface
    *ppEnum=(LPENUMRECT)IMPIEnumRect_Constructor();

    if (NULL==*ppEnum)
        return FALSE;

    //If creation worked, AddRef the interface
    if (NULL!=*ppEnum)
        (*ppEnum)->lpVtbl->AddRef(*ppEnum);

    return (NULL!=*ppEnum);
    }

LPIMPIENUMRECT IMPIEnumRect_Constructor(void)
    {
    LPIMPIENUMRECT      pER;
    UINT                i;
```

(continued)

Listing 3-3. *continued*

```
/*
 * First time through initialize function table.  Such a table
 * could be defined as a constant instead of doing explicit
 * initialization here.  However, this method shows exactly
 * which pointers are going where and does not depend on knowing
 * the ordering of the functions in the table, just the names.
 */

    if (!fVtblInitialized)
        {
        vtEnumRect.AddRef =IMPIEnumRect_AddRef;
        vtEnumRect.Release=IMPIEnumRect_Release;
        vtEnumRect.Next   =IMPIEnumRect_Next;
        vtEnumRect.Skip   =IMPIEnumRect_Skip;
        vtEnumRect.Reset  =IMPIEnumRect_Reset;

        fVtblInitialized=TRUE;
        }

    pER=(LPIMPIENUMRECT)_fmalloc(sizeof(IMPIENUMRECT));

    if (NULL==pER)
        return NULL;

    //Initialize function table pointer
    pER->lpVtbl=&vtEnumRect;

    //Initialize the array of rectangles
    for (i=0; i < CRECTS; i++)
        SetRect(&pER->m_rgrc[i], i, i*2, i*3, i*4);

    //Ref counts always start at zero
    pER->m_cRef=0;

    //Current pointer is the first element.
    pER->m_iCur=0;

    return pER;
    }

void IMPIEnumRect_Destructor(LPIMPIENUMRECT pER)
    {
    if (NULL==pER)
        return;
```

(continued)

Listing 3-3. *continued*

```
    _ffree((LPVOID)pER);

    return;
    }

DWORD IMPIEnumRect_AddRef(LPENUMRECT pEnum)
    {
    LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;

    if (NULL==pER)
        return 0L;

    return ++pER->m_cRef;
    }

DWORD IMPIEnumRect_Release(LPENUMRECT pEnum)
    {
    LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;
    DWORD               cRefT;

    if (NULL==pER)
        return 0L;

    cRefT=--pER->m_cRef;

    if (0==pER->m_cRef)
        IMPIEnumRect_Destructor(pER);

    return cRefT;
    }

BOOL IMPIEnumRect_Next(LPENUMRECT pEnum, DWORD cRect
    , LPRECT prc, LPDWORD pdwRects)
    {
    LPIMPIENUMRECT  pER=(LPIMPIENUMRECT)pEnum;
    DWORD           cRectReturn=0L;

    if (NULL==pdwRects)
        return FALSE;

    *pdwRects=0L;

    if (NULL==prc || (pER->m_iCur >= CRECTS))
        return FALSE;
```

(continued)

Listing 3-3. *continued*

```
    while (pER->m_iCur < CRECTS && cRect > 0)
        {
        *prc++=pER->m_rgrc[pER->m_iCur++];
        cRectReturn++;
        cRect--;
        }

    *pdwRects=(cRectReturn-cRect);
    return TRUE;
    }

BOOL IMPIEnumRect_Skip(LPENUMRECT pEnum, DWORD cSkip)
    {
    LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;

    if (NULL==pER)
        return FALSE;

    if ((pER->m_iCur+cSkip) >= CRECTS)
        return FALSE;

    pER->m_iCur+=cSkip;
    return TRUE;
    }

void IMPIEnumRect_Reset(LPENUMRECT pEnum)
    {
    LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;

    if (NULL==pER)
        return;

    pER->m_iCur=0;
    return;
    }
```

When IENUM0.H is compiled for C++, it generates a C++ abstract base class—that is, a base class that defines a set of pure virtual functions (by using *virtual* and *=0*). In addition, IENUM0.H defines a far pointer type for this interface in the conventional form LP<INTERFACE>, where <INTERFACE> is the interface name in all caps excluding the *I* prefix. The C++ implementation of the *RECTEnumerator* object, *CImplEnumRECT* in ENUMCPP.H, inherits these function signatures from this interface and provides each implementation. Instantiating this C++ class will generate an IEnumRECT function table for you.

Defining an interface in C is more work, primarily because you have to construct the function table manually. In ENUMC, the structure *IEnumRECT-Vtbl*[5] is a structure of function pointers that is exactly what many C++ compilers create internally for C++ classes. The actual interface, *IEnumRECT*, is defined as a structure that contains a pointer to this function table. So when a C application has a pointer to an interface, it really has a pointer to a pointer to a function table. The C implementation of the *RECTEnumerator* object, a structure named IMPIENUMRECT in ENUMC.H, duplicates the *lpVtbl* member of *IEnumRECT* in its own structure, thereby making a pointer to IMPIENUM-RECT polymorphic with a pointer to *IEnumRECT*. This duplicates what happens automatically with C++ classes and is common in C-based OLE 2 code.

Creating the *RECTEnumerator* Object

When you choose the Create command from the Enum menu of either ENUMC or ENUMCPP, you generate a call (see IDM_ENUMCREATE in the WM_COMMAND *switch*) to *CreateRECTEnumerator*. This creation function creates the object and returns the *IEnumRECT* interface pointer in an *out-parameter*—that is, the caller passes the address in which *CreateRECTEnumerator* stores the interface pointer. This technique is used everywhere in OLE 2 to allow standardization of almost all return values into a type named HRESULT, described in the later section "HRESULT and SCODE." To avoid further complicating your life now with HRESULT, we'll stick with a BOOL return type.

Note that specific functions to create specific types of object are rare in OLE 2. Most often you use a class factory object (and *IClassFactory::CreateInstance*), which eliminates the need for most, but not all, API functions such as *CreateRECTEnumerator*. As we'll see in Chapter 4, implementations of *IClassFactory::CreateInstance* look very much like *CreateRECTEnumerator*, in either language.

In this example, the ENUMC and ENUMCPP programs are both user and implementor of the same object. Both programs use an internal function, *CreateRECTEnumerator*, to obtain an *IEnumRECT* pointer to the *RECTEnumerator* object. This demonstrates the typical fashion through which a user obtains an interface pointer by calling an API function. Internally, the *CreateRECTEnumerator* creates the function table for the *IEnumRECT* interface and then allocates and initializes the object itself.

In C++, the *new* operator applied to the *CImplEnumRECT* class automatically allocates memory for the object and creates the function table. All you have to do is initialize the object. Because the *CImplEnumRECT* class

5. The jargon name *Vtbl* means *virtual function table*, which is always referred to in this book as simply a function table.

inherits from *IEnumRECT*, you can typecast the pointer from *new* to an *LPENUMRECT*, which turns it into a pointer to only the interface. So in C++, it's highly convenient to implement objects as C++ objects, although the pointers you return are always interface pointers.

In C, you must manually fill the function table, manually allocate the object's memory, and then initialize it exactly as in C++. The function *IMPIEnumRECT_Constructor* handles all this for you and lets you use it in place of the *new* operator in C++. This constructor function first creates the function table by storing function pointers in a global array of type *IEnumRECTVtbl*. (This needs to happen only once for all instances of the object.) Only the implementation of this object knows that the function table actually exists in a global variable such as this. The object's user sees only the table but has no knowledge about where that table lives. In any case, *IMPIEnumRECT_Constructor* then allocates the object's structure and stores a pointer to the function table in the *lpVtbl* member of the object. Finally, it performs the same initialization as in C++.

Using an *IEnumRECT* Pointer

You will also notice that *CreateRECTEnumerator* calls the object's *AddRef* function before returning the pointer. This rule of reference counting, one of several, is explained in "Reference Counting." However, this one call, along with calls to the other *IEnumRECT* functions in ENUMC and ENUMCPP, demonstrates the calling differences between C and C++. Given an interface pointer, a C++ user calls member functions through the pointer as with any other C++ object pointer:

```
//C++ call to interface member function
pIEnumRect->AddRef();
```

This will land in *CImpIEnumRECT::AddRef* just as any other C++ call would. The *this* pointer inside the member function is identical to *pIEnumRECT*, through which the function was called.

In C, we have a more complicated story. First, any member function call made through an interface pointer must be indirect through the *lpVtbl* member before it gets at the function (an indirection done automatically in C++):

```
//C call to interface member function
pIEnumRect->lpVtbl->AddRef(pIEnumRect);
```

To ensure that the implementation of *IEnumRECT::AddRef* invoked here knows which object is being accessed, C users must pass the same interface pointer as the first parameter to the function. This mimics the behavior of

the *this* pointer that is automatic in C++. Because this extra parameter is necessary, the function prototypes in IENUM0.H for the C interface had to include the pointer type as the first parameter.

The two lines of preceding code illustrate why C++ is more *convenient*, but no more *functional*, than C in using Windows Objects. The C user will always need the extra indirection and the extra parameter, which can quickly add up to a lot of extra code. By no means, however, does that small fact render it impossible to write C code for OLE 2. An object implementor in C needs only to provide for creating the function table manually. But aside from these few differences, programming in OLE 2 is identical in either language.

Reference Counting

The implementation of the *RECTEnumerator* object is illustrative, but not useful. To build the bridge between illustrative objects and useful objects, we need more information that applies to all the remaining chapters of this book. One of the most important subjects is reference counting, which is a set of rules that control an object's lifetime.

If you are an object, your reference counting requires that you live a unique life in which you are not allowed to rest eternally unless all your acquaintances have also passed on. (I say "unique life" because if everyone lived this way, we'd all be immortal. But an object user does not have a reference count and therefore does not live such a life.) At birth, you form an acquaintance with your mother, and as you live your life, you meet new people and form new acquaintances. Whenever you form a new relationship, you increment your reference count. Whenever an acquaintance dies, you are released of that relationship, and you decrement your reference count. Only when all such relationships end are you allowed your personal journey to the afterlife. That means that for you, the object, your reference count is zero, and you are allowed to free your memory.

You might have noticed a potential problem with this. If you are an object as well as an object user, and the object you are using just so happens to be the user that is using your object, you have a seeming paradox of mutual immortality on your hands. Neither party can die because each has an acquaintance with the other, or what is known as a circular reference count. In such cases...free the other object, as well. In other words, the Almighty End User strikes one object down with a lightning bolt from the sky, thereby breaking the circular reference. In such cases, you have to remember the Almighty End User, who does things such as close an application. This act overrides the

relationship rule; the object in the application that's closing brutally terminates all connections to it in such a way that its reference count is reduced to zero, which might free the other object as well.

The rules governing reference counting can be distilled into two fundamental principles:

- Creation of a new interface pointer to an object must be accompanied by an *AddRef* call to the object through that new pointer.
- Destruction of an interface pointer (that is, when the pointer goes out of scope) must be accompanied by a *Release* call through that pointer before it can be destroyed.

This means that whenever you assign one pointer to another in some piece of code, you should use *AddRef* for the new copy (the left operand) of the pointer. Before that pointer is overwritten, it must have *Release* called for it. All *AddRef* and *Release* calls made through interfaces affect the reference count of the entire object, which is shared among all interfaces on that object. Consider the following code:

```
LPSOMEINTERFACE    pISome1;
LPSOMEINTERFACE    pISome2
LPSOMEINTERFACE    pCopy;

//A function that creates the pointer uses AddRef on it.
CreateISomeObject(&pISome1);   //Some1 ref count=1
CreateISomeObject(&pISome2);   //Some2 ref count=1

pCopy=pISome1;                 //Some1 count=1
pCopy->AddRef();               //AddRef new copy, Some1=2

[Do things]

pCopy->Release();             //Release before overwrite, Some1=1
pCopy=pISome2;                //Some2=1
pCopy->AddRef();              //Some2=2

[What kinds of things do you do]

pCopy->Release();             //Release before overwrite, Some2=1
pCopy=NULL;

[Things that make us go]

pISome2->Release();    //Release when done, Some2=0, Some2 freed.
pISome1->Release();    //Release when done, Some1=0, Some1 freed.
```

An object's lifetime is controlled by all *AddRef* and *Release* calls on all its interfaces combined. Reference counting for a specific interface is useful in debugging to verify that your user is counting properly, but it is the object reference count that matters. According to the first fundamental principle of reference counting, any function that returns a pointer to an interface must call *AddRef* through that pointer. Functions that create an actual object and return the first pointer to an interface on that object are such functions, like *CreateISomeObject* in the preceding example. Now anytime you create a new copy of a pointer, you must also call *AddRef* through that new copy because you have two different references, two different pointer variables, to the same object that are independent. Then according to the second principle of reference counting, all *AddRef* calls must be matched with a *Release* call. So before your pointer variables are destroyed (by an explicit overwrite or by going out of scope), you must call *Release* through that pointer. This includes calling *Release* through any pointer copy (through which you called *AddRef*) as well as through the pointer you obtained from the function that created the object. Functions that create objects and return interface pointers are the functions that actually create the pointers. Such functions fill the *out-parameters* from which the caller receives the pointer. Therefore, it is the creator, not the caller, that is responsible for the first *AddRef* on the object by means of the interface pointer it initially returns.

My Kingdom for Some Optimizations!

The stated rules and their effect on the code shown earlier probably seem rather fascist. Well, they are, but that doesn't mean there's no underground movement.

When you know the lifetimes of all interface pointers to the same object, you can bypass the majority of *AddRef* and *Release* calls. There are two manifestations of such knowledge: nested lifetimes and overlapping lifetimes.

In the preceding code, every instance of *pCopy* is nested within the lifetimes of *pISome1* and *pISome2*—that is, the copy lives and dies within the lifetime of the original. After *CreateISomeObject* is called, both objects have a reference count of one. The lifetimes of their pointers is bounded by these create calls and the final *Release* calls made through those pointers. Because we know these lifetimes, we can eliminate any other *AddRef* and *Release* calls to copies of those pointers:

```
LPSOMEINTERFACE    pISome1;
LPSOMEINTERFACE    pISome2;
LPSOMEINTERFACE    pCopy;
```

```
CreateISomeObject(&pISome1);    //Some1 ref count=1
CreateISomeObject(&pISome2);    //Some2 ref count=1

pCopy=pISome1;                  //Some1=1, pCopy nested in Some1's life

[Do things]

pCopy=pISome2;                  //Some2=1, pCopy nested in Some2's life

[Do other things]

pICopy=NULL;                    //No Release necessary

[Do anything, then clean up]

pISome2->Release();     //Release when done, Some2=0, Some2 freed.
pISome1->Release();     //Release when done, Some1=0, Some1 freed.
```

Overlapping lifetimes are those in which the original pointer dies after the copy is born but before the copy itself dies. If the copy is alive at the original's funeral, it can inherit ownership of the reference count on behalf of the original:

```
LPSOMEINTERFACE    pISome1;
LPSOMEINTERFACE    pCopy;

CreateISomeObject(&pISome1);    //Some1 ref count=1

pCopy=pISome1;     //Some1=1, pCopy nested in Some1's life
pISome1=NULL;      //Pointer destroyed, pCopy inherits count, Some1=1

pCopy->Release(); //Release inherited ref count, Some1=0, Some1 freed.
```

With these optimizations, reference counting can be reduced to four specific rules, in which an *AddRef* for a new copy of a pointer is necessary (and thus must have a *Release* call made through it when destroyed):

- Functions that return a new interface pointer in an *out-parameter* or as a return value must call *AddRef* for the object through that pointer before returning.
- Functions that accept an *in-out parameter* must call *Release* for the *in-parameter* before overwriting it and must call *AddRef* for the *out-parameter*. Callers of these functions must call *AddRef* for the passed pointer to maintain a separate copy if the function is known to call *Release* for that pointer.
- If two pointers to the same object have unrelated lifetimes, *AddRef* must be called for each.
- Call *AddRef* for each local copy of a global pointer.

In all cases, some piece of code must call *Release* for every *AddRef* on a pointer. In the first of the preceding cases, the caller of a function that returns a new pointer (such as *CreateISomeObject*) becomes responsible for that new object. When the caller has finished with the object, it must call *Release*. If the object's reference count is decreased to zero because of this, the object may be destroyed at the discretion of the implementor, but from the user's point of view, the object is gone. If you fail to use *Release* for a reference count, you generally doom the object to the boredom of useless immortality—memory might not be freed, or the DLL or EXE supplying that object might not unload. Be humane to your objects: Be sure to release them.

Call-Use-Release

The first optimized reference counting rule exposes a common pattern in OLE 2 programming. To use an object, you will call some function that returns a pointer to an interface. That function will call *AddRef* on behalf of this new pointer. You then use that pointer for as long as you want. When you have finished with it, you call *Release* through that pointer to let the object know you no longer need it.

The same object might, in fact, be in use through other pointers, even in another process. As far as you're concerned, you call *Release* to free the reference count for which you are responsible, and you know that after that time you cannot access that object again because it might have freed itself. If there are other outstanding pointers to that object elsewhere, however, the object is still in memory, but you are oblivious to that fact.

This pattern, which I refer to as *Call-Use-Release*, is common in OLE 2 programming. There are many functions you call to obtain pointers with a reference count, and there are many different things to do with those pointers (which is why this book is so thick). But regardless of how you got the pointer or what you did with it, you must call *Release* through it when you are finished.

The final *Release* can do more than simply free the object—"free the object" can imply many other actions. For example, Compound File objects discussed in Chapter 5 might close a file; a memory manager object we'll see in Chapter 4 will free any allocations it has made; a compound document object we'll implement in Chapter 10 might close down an application. Because the *Release* member function can be overloaded in this manner, you will notice an absence of "close" API functions in OLE 2. There is a function to open a Compound File, but there is no function to close it—the API provides the initial *AddRef*, and closure is handled in the final *Release*.

IUnknown, the Root of All Evil

From the preceding discussion, you can isolate two fundamental interface and object operations: reference counting and pointer creation. The interface named *IUnknown*, which all Windows Objects support, encapsulates these two ideas in three member functions:

Function	Result
QueryInterface	Returns a pointer to the requested interface on the same object. *QueryInterface* is considered a function that creates a pointer, so it calls *AddRef* through any pointer it returns.
AddRef	Increments the object's reference count, returning the current count.
Release	Decrements the object's reference count, returning the new count, and can free the object when the reference count reaches zero.

Because *AddRef* and *Release* behave exactly as described in the previous section, we won't examine them further here. Instead, we'll look more closely at *QueryInterface*.

QueryInterface is more than simply the fundamental creator of interface pointers, although it does always return a pointer to a different interface on the same object. *QueryInterface* allows you to access each separate function table supported by an individual object. How you obtain the *first* interface pointer on the object is one thing—*QueryInterface* allows you to get to all other interface pointers on the same object *after* creation.

QueryInterface allows an object user to discover an object's capabilities at runtime, instead of having to incorporate specific knowledge about objects at compile-time. You learn capabilities by asking for additional interfaces that the object supports, a process called *interface negotiation*. When you create an arbitrary object, you'll always get back an interface pointer that looks like an *IUnknown* pointer because all other interfaces incorporate *IUnknown*. So if you are able to get an object, you can *always* call *QueryInterface*.

With an *IUnknown* pointer, you can now determine whether the object supports a particular feature by calling *QueryInterface*. For example, to determine whether the object supports data transfer, call *QueryInterface* asking for an *IDataObject* interface (see Chapter 6). To determine whether the object is a Compound Document object, meaning that it can be treated in a standard way for editing capabilities, call *QueryInterface* for *IOleObject* (see Chapters 9

and 10). *IOleObject* describes only an embedded object, so if you want to determine whether it supports linking, call *QueryInterface* for *IOleLink* (see Chapter 12). To go even further, you can ask the object whether it supports in-place activation by calling *QueryInterface* for *IOleInPlaceObject*. (See Chapters 15 and 16.)

When you call *QueryInterface* for a new pointer, you not only learn whether the object is capable of the set of functions implied by that interface, but you receive back the interface pointer through which you access those functions. This means that you cannot possibly attempt to use certain features of an object if it does not support those features because you can never get the appropriate interface pointer from the object. In other words, if you speak a different language than I do, we cannot communicate; only when we establish a common language can we express our ideas (functions) to one another. Furthermore, it is impossible for me to offend you verbally unless I speak in your language; or in Windows Objects terms, I am not able to pass the wrong object to a function that does not understand that object because I must use the language of the object to perform any function on it.

Applications benefit from being able to make decisions dynamically about how to treat an object based on that object's capabilities, instead of rigidly compiling such behavior. Let's say I work at the United Nations in New York City and I speak English and German. I walk into a room with 10 international delegates with whom I need to discuss a few issues. I go up to one of the delegates and ask "Do you speak English?" This query is met with an affirmative, "Yes." Great, now we can talk. Partway through our conversation, I find that I simply cannot express one of my ideas in English, but I know I could express it in German—some languages have words without equivalents in other languages. So I ask, "Sprechen Sie Deutsch?" to which the other person responds "Ja." Because my partner also speaks German, I can now express my idea in that language. If German were not in my partner's repertoire, we would be limited to speaking English only.

My ability to communicate with anyone is limited not by the number of languages I speak, but by the languages the other person and I have in common. This means that my level of communication varies from person to person. With some people, I can converse in two languages; with others, I might converse in only one, or I might not be able to converse at all. The key points are that I learn this when I meet the person and that my knowing many languages allows me to speak with many more people, not just with people who speak exactly the same set of languages.

How does this apply to objects? The *QueryInterface* mechanism allows an object, or a user of objects, to implement or to be able to use as many interfaces as desired without any fear of restricting your ability to use objects or be used by some object user. For example, a compound document object can implement full in-place activation capabilities without restricting itself to being useful only to in-place container applications. A non-in-place container can still use that object as a non-in-place object, and in such a case, the in-place activation interfaces are ignored entirely.

I've often been asked why there is not a function that returns a list of all the interfaces an object supports. The answer is that such a function would be, for the most part, useless. What would you do programmatically with such information? Although it might be useful in some very esoteric circumstances, you never really need to know whether an object supports an interface unless you intend to perform some function through that interface. So you ask for it via *QueryInterface*. Furthermore, the list of interfaces that an object of a specific class supports is constant only within a specific object's lifetime and might vary between different instantiations of objects of the same class. Therefore, you cannot assume that if Object 1 of class X supports these interfaces, Object 2 of class X does as well. You must also not assume that if objects of class X once supported interface Y, they always will, because the object might change in the meantime. Because having some list of interfaces doesn't get you far and because the capability to obtain such a list is outright dangerous, *QueryInterface* is the only way to learn about an object's capabilities.

QueryInterface vs. Inheritance

Use of *QueryInterface* is superior to use of C++ base classes and C++ inheritance for two reasons. First, given an arbitrary C++ object pointer to some base class object, you really have no way to determine whether that pointer is actually referring to some derived class object instead—you have no way to examine the virtual function table to see exactly what kind of object you have. Therefore, you are *always and forever* restricted to dealing with that object on the base class's terms. Using *QueryInterface*, on the other hand, allows you to get at any function table you want from the base *IUnknown* interface. Given any *IUnknown*, you can find out how rich the object actually is. You can get from the base to more specific interfaces.

The second major advantage of *QueryInterface* is that unless an object supports an interface, you cannot call member functions that the object does not support. This is not true of C++ objects. Take, for example, the base object class *CObject* in Microsoft's Foundation Classes. A *CObject* might be capable of

serializing itself to a storage device (such as a file); to ask the question "Can you serialize yourself?", you call the member function *IsSerializable.* If the answer is positive, you can then call another function, *Serialize,* to actually perform the task.

That's nice, but a user of a *CObject* is in no way barred from calling *Serialize* at any time. In other words, serialization capability is not tightly coupled to the question of whether the object can actually serialize. It is therefore possible to call *Serialize* on an object that does not support it, with unpredictable results. The *QueryInterface* mechanism, on the other hand, does tightly couple the question and the capability. You must ask the object via *QueryInterface* whether it supports a particular functionality, and only if it does are you provided the interface through which to call such functions. Given an arbitrary *IUnknown* object, you cannot possibly ask it to serialize itself without first asking for an interface that knows about serialization. If the object does not support the capability, you cannot get the interface. Therefore, you cannot call unsupported functions, and you eliminate the possibility of unpredictable behavior.

QueryInterface Properties and Interface Lifetimes

There are a number of rules in the OLE 2 Design Specifications concerning the behavior of *QueryInterface.* The first and most important rule is that any call to *QueryInterface* asking for *IUnknown* through any interface on the object must always return the *exact same pointer value.* The specific reasoning for this is that given two arbitrary interface pointers, you can determine whether they belong to the same *object* by asking each for an *IUnknown* pointer and comparing the actual pointer values. If they match, application of this rule allows both interface pointers refer to the same object.

The second rule is that after an object is instantiated, the interfaces it supports are static. This means that if *QueryInterface* succeeded for a particular interface at one point in the object's lifetime, an identical call to *QueryInterface* at a later time will also work. This does not mean that the exact pointer *values* returned from both calls will be identical—it means only that the interface is always available. Note that the static set of available interfaces applies to a specific object instantiation, not an object class—that is, two objects of the same type might not both support the same interfaces, but during the lifetime of each, the interfaces they each support will remain static.

The third rule is that the *QueryInterface* operation must be reflexive, symmetric, and transitive, as described in the following table (in which *IInterface1, Interface2,* and *Interface3* are hypothetical):

QueryInterface Property	Meaning
Reflexive	*pInterface1->QueryInterface(IInterface1)* must succeed.
Symmetric	If *pInterface2* was returned from *pInterface1->QueryInterface(IInterface2)*, then *pInterface2->QueryInterface(IInterface1)* must also succeed.
Transitive	If *pInterface2* was obtained from *pInterface1->QueryInterface(IInterface2)* and *pInterface3* was obtained from *pInterface2->QueryInterface(IInterface2)*, then *pInterface3->QueryInterface(IInterface1)* must succeed.

In all these cases, "must succeed" is not so strong as to imply that these cannot fail under the most catastrophic situations. In addition, these properties do not mean that the same pointer *value* is always returned for the interface, with the exception of *IUnknown.*

The final rule has to do with the lifetime of a particular *interface pointer,* as opposed to the lifetime of the entire object. The rule is that as long as the object is alive, all interface pointers obtained on that object must remain valid, even if the *Release* function has been called through those pointers. Consider the following code:

```
LPSOMEINTERFACE     pSome;
LPOTHERINTERFACE    pOther;

CreateSomeObject(&pSome);                     //Object ref count is 1
pSome->QueryInterface(IOtherInterface, &pOther);  //ref count is 2
pOther->Release();                                //ref count is 1
/*
 * Since the object is still alive, pOther is still a valid
 * interface pointer although Release has been called through it.
 */

pSome->Release();           //ref count is 0, object destroyed.

//pSome and pOther are now invalid.
```

When we first obtain the *pSome* pointer, the object will have a reference count of one. The object will therefore remain alive as long as the reference count remains above zero. When we query for *pOther,* the object will have a reference count of two. When we call *pOther->Release* the object will still have a positive reference, meaning that *pOther* will still be valid, even though we called *Release* through it. That is, we can still call member functions through *pOther.* This is because the interface is alive as long as the object is alive. Only

when we call *pSome->Release* and reduce the object's reference count to zero will the object be destroyed, thus invalidating all interface pointers on that object.

In later chapters, we'll see a few circumstances in which this rule becomes important. For now, this illustrates why interface-level reference counting is useful only for debugging purposes. A zero reference count on an interface means neither that the interface is invalid nor that the object is invalid. The only important reference count is the one on the entire object, which all implementations of *AddRef* and *Release* on all interfaces of that object must return.

Some Data Types and Calling Conventions

If you look in OLE 2's include file, COMPOBJ.H, you will find *IUnknown* declared as follows:

```
DECLARE_INTERFACE(IUnknown)
    {
    STDMETHOD(QueryInterface) (THIS_ REFIID riid
        , LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG,AddRef) (THIS)  PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;
    };
```

Offhand, this might look very odd, but there are a number of macros shown here that are used in many other aspects of OLE 2.

DECLARE_INTERFACE, STDMETHOD, STDMETHOD_, THIS, THIS_, and PURE are all macros that hide the differences between C and C++ interface definitions as well as those among Win16, Win32, and Macintosh implementations. When this interface declaration is compiled in C++, the result is similar to the definition of *IEnumRECT* in IENUM0.H; the same goes for a C compilation. For complete details about how these macros expand, see the comments in the COMPOBJ.H file in your OLE 2 SDK.

Note also that interfaces shown in the ENUMC program are not exactly what is generated through these macros because real OLE 2 interfaces differ in calling convention and return type. The following sections look at these, as well as the REFIID type, in more detail.

STDMETHOD and Associates

The STDMETHOD macro expands into HRESULT STDMETHODCALLTYPE. HRESULT is a special return value type discussed in the next section, "HRESULT and SCODE." STDMETHODCALLTYPE is defined under Microsoft Windows 3.1 as *__export __far __cdecl* and under Windows NT as

__export __cdecl. The *cdecl* type was necessary to support generation of the proper stack frame for member function calls, portability of C code from 16-bit to 32-bit, and interoperability between C and C++ implementations. The OLE 2 architects would have preferred to have used a more efficient calling convention (such as PASCAL) but were unable due to these constraints.

The STDMETHOD_ (<type>) macro allows a variation on the return type from HRESULT into any other type. *AddRef* and *Release*, for example, return the new reference count instead of an HRESULT. In this sense, they are fairly unique: Most other interface members in all of OLE 2 return HRESULTs, including *QueryInterface*.

Under C++, both STDMETHOD macros include the *virtual* keyword. The PURE macro also compiles under C++ as *=0* to generate pure virtual members in the interface declarations. Of course, C compilations include neither (pure compiles to nothing).

Because the two possible STDMETHOD macros generate the *virtual* and *=0* signatures, they are not used when implementing interface functions, only when declaring one. Instead, you use STDMETHODIMP or STDMETHODIMP_(<type>), which does nothing more than eliminate the *virtual* keyword in C++ compilations but still generate either an HRESULT or a <type> return value along with STDMETHODCALLTYPE.

HRESULT and SCODE

OLE 2 introduces a new return type used by *QueryInterface* and almost every other interface member function: HRESULT, or handle to a result. Conceptually, an HRESULT is a *status code*, or *SCODE*, that describes what occurred and a handle that can be used to obtain additional information about an error or how to recover from it. The intention is that over time, interfaces will return very detailed information that can describe a suggested course of action when failure occurs.

The HRESULT and SCODE types are both 32-bit values containing a severity flag, a facility code, and an information code, as shown in Figure 3-9 on the next page. The Context field is what distinguishes an HRESULT from an SCODE because the Context is always zero in an SCODE but it could contain a handle to additional information in an HRESULT.[6] An SCODE is created or dissected with various macros in the OLE 2 include file SCODE.H. such as MAKE_SCODE. Some of the more commonly used SCODE values are shown in Table 3-1 on the next page. The Facility field of an SCODE

6. In OLE 2, the Context field in an HRESULT is always zero, but this will change in the future.

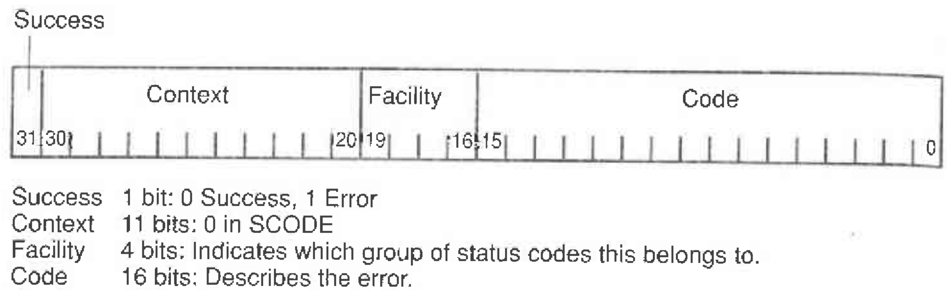


Figure 3-9.
Structure of an HRESULT and an SCODE.

describes the source of the error, which might be in the marshaling of the function call, in the interface function itself, or elsewhere. Any SCODE prefixed with *S_* carries information and means *success*, whereas an SCODE prefixed with *E_* means *failure* and carries a code describing the failure. Many other SCODE symbols defined in the OLE 2 header files are prefixed with other labels, such as *OLE_*, to identify the specific subtechnology generating the error. *OLE_*, for example, means Compound Documents.

Value	Meaning
S_OK	Function succeeded. Also used for functions that semantically return boolean information that succeeds with a TRUE result.
S_FALSE	Function that semantically returns boolean information that succeeds with a FALSE result.
E_NOINTERFACE	*QueryInterface* could not return a pointer to the requested interface.
E_NOTIMPL	Member function contains no implementation.
E_FAIL	Unspecified failure.
E_OUTOFMEMORY	Function failed to allocate necessary memory.

Table 3-1.
Common SCODE values.

Because the HRESULT and SCODE types are not straight equivalents, OLE 2 provides a few functions (implemented in version 2 as macros) that provide conversions between an HRESULT and an SCODE, both of which you'll use often in your own implementations. To create an HRESULT from an SCODE, use the function *ResultFromScode(SCODE)*. To dig an SCODE out of an HRESULT, use the function *GetScode(HRESULT)*, which is also a macro.

Although this seems like a pain, especially on the receiving end of an HRESULT, the most common case allows you to bypass these functions altogether. If a function works completely, it can return the predefined HRESULT called NOERROR, the equivalent of an HRESULT containing S_OK. The code receiving the HRESULT can use one of two macros to determine the success or failure of the function (*hr* stands for an HRESULT):

Macro	Result
SUCCEEDED(*hr*)	Tests the high bit of the HRESULT and returns TRUE if that bit is clear. This will return TRUE for any S_ SCODE and FALSE for any E_ SCODE.
FAILED(*hr*)	Tests the high bit of the HRESULT and returns TRUE if that bit is set. This will return TRUE for any E_ SCODE and FALSE for any S_ SCODE.

Using SUCCEEDED and FAILED is preferred to comparing an HRESULT to NOERROR directly because some codes, such as S_FALSE or STG_S_CONVERTED (see Chapter 5) mean that the function actually succeeded and is returning more information than that simple fact. A test such as *(NOERROR!=hr)* will be TRUE when an HRESULT contains S_FALSE, whereas the *FAILED(hr)* macro will be FALSE. When a function returns either the S_OK code or S_FALSE code, you *should* compare the HRESULT to NOERROR because a macro such as SUCCEEDED will return TRUE for both codes. The *GetScode* function is necessary only when you want to find the exact reason for failure instead of simply the fact that the function did fail.

Globally Unique Identifiers: GUIDs, IIDs, CLSIDs

Every interface is defined by an *interface identifier*, or IID (as in IID_IUnknown), which is a special case of a universally unique identifier, or UUID. The universally unique identifier is also known as the globally unique identifier, or GUID (pronounced *goo-id*). GUIDs are 128-bit values created with a DEFINE_GUID macro (see INITGUID.H in the OLE 2 SDK). Every interface and object class uses a GUID for identification. As described in the OLE 2 SDK, Microsoft will allocate one or more sets of 256 GUIDs for your exclusive use when you request them, or if you have a network card in your machine, you can run a tool named UUIDGEN.EXE that will provide you with a set of 256 GUIDs based on the time of day, the date, and a unique number contained in your network card. The chance of this tool generating duplicate GUIDs is about the same as two random atoms in the universe colliding to form a small avocado. In other words, don't worry about it.

All the code shown in this book uses GUIDs prefixed with *000211*, which are allocated to the author. Do not use these GUIDs for your own products.

OLE 2 defines IIDs for every standard interface along with class identifiers (CLSID) for every standard object class. When we call any function that asks for an IID or a CLSID, we pass a *reference* to an instance of the GUID structure that exists in our process space using the types REFIID or REFCLSID. When passing an IID or a CLSID in C, you must use a pointer—that is, pass *&IID_** or *&CLSID_**, where REFIID and REFCLSID are typed as *const* pointers to IID or CLSID. In C++, because a reference is a natural part of the language, you drop the &. We will see more specifics about the definition and use of GUIDs in Chapter 4 and beyond.

Finally, to compare two GUID, IID, or CLSID values for equality, use the *IsEqualGUID*, *IsEqualIID*, and *IsEqualCLSID* functions defined in COMPOBJ.H. There the latter two are simply more readable aliases for *IsEqualGUID*. If you are programming in C++, take a look at COMPOBJ.H, which defines an overloaded "==" operator for the GUID type that, of course, applies equally well to the IID and CLSID types. In this book, I'll use the appropriate *IsEqual...* function to keep the code more usable for C programmers.

OLE 2 Interfaces and API Functions

OLE 2 defines no fewer than 62 interfaces, many of which it implements and uses internally. Those of importance to applications are shown in Figure 3-10, grouped by the technology area to which they apply. Remember that higher technologies build on the lower technologies, as discussed in Chapter 1.

This picture might look a little intimidating at first. Many interfaces are shown, but applications have to implement only a handful. For the basic compound document container applications we'll see in Chapter 9, you need to implement only *IOleClientSite* and *IAdviseSink*. Although you implement only a few, you *use* many more, thereby contributing to the magnitude of Figure 3-10. In addition, some of the interfaces shown are useful only as base interfaces for others. Rarely, if ever, will you use or implement a simple base class by itself.

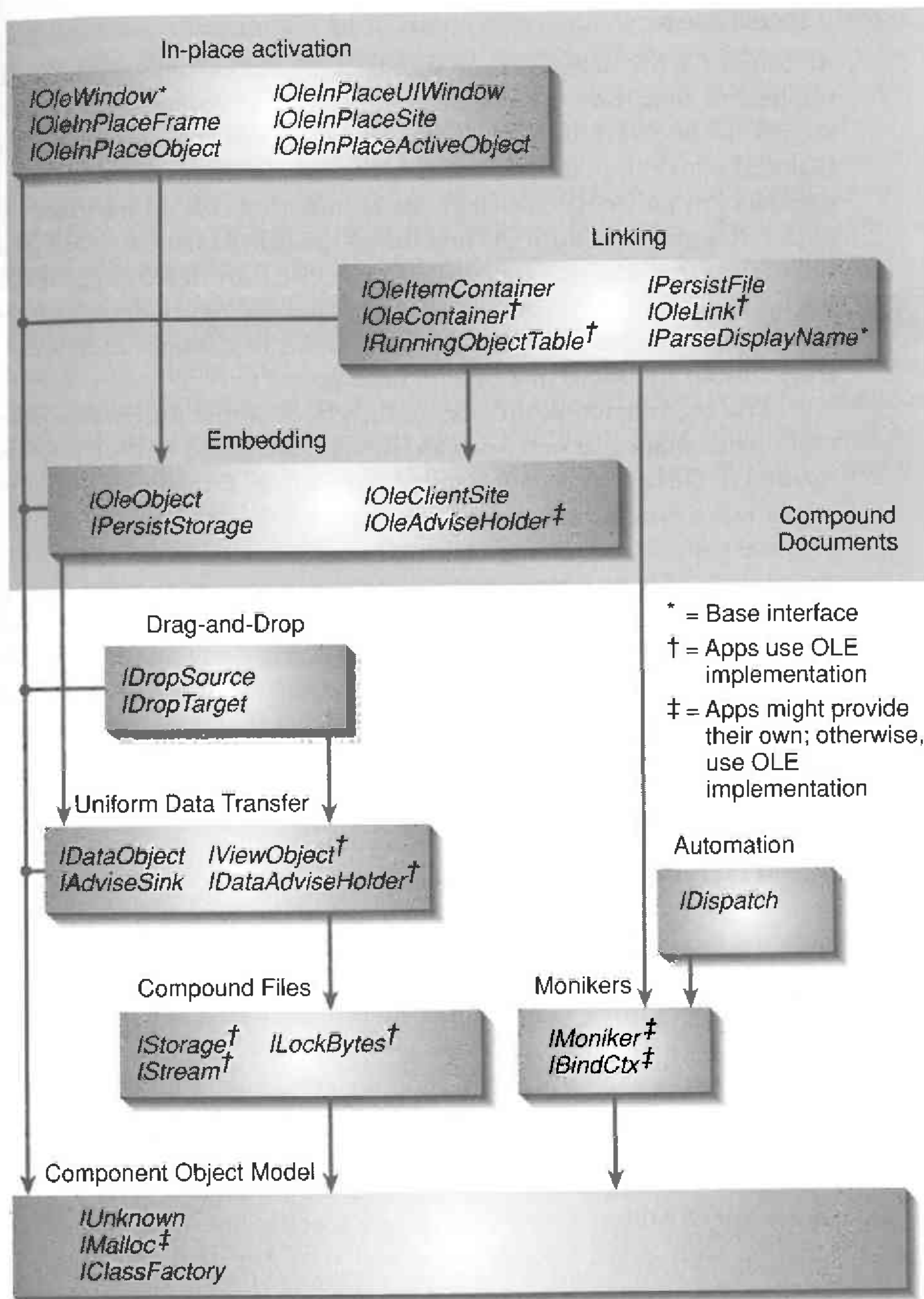


Figure 3-10.
Interesting interfaces in OLE 2 technologies.

Custom Interfaces

Although OLE 2 defines many standard interfaces, objects can define and implement their own interfaces as long as their potential users are implemented to be aware of those interfaces. To reiterate a point, the interfaces that make up compound documents aim to eliminate the need for custom interfaces in particular scenarios. By eliminating custom interfaces from an object, you greatly reduce the amount of specialized code needed in a potential user of that object. The Polyline object we start developing in Chapter 4 begins its life with a custom interface, but throughout this book we'll convert many pieces of it to use standard Compound Document interfaces, leaving truly custom functions in a custom interface.

The big restriction on custom interfaces is that unless you provide for custom marshaling as well, you can use such interfaces only on objects implemented in DLLs that do not require marshaling. Providing custom marshaling is not a simple task, and I recommend that you wait until Microsoft provides generic marshaling code so that all you need to do is provide it with a description of your function parameters, their types, and the way in which they are marshaled. If you absolutely must use the capability now, study the information in the OLE 2 SDK, primarily that concerning the *IMarshal* interface and related functions. Custom marshaling is not covered in this book.

Interfaces vs. API Functions

In developing with OLE 2, you'll soon notice that you use relatively few API functions to achieve your goals. Instead, you call many interface functions. Almost everything an OLE 2 application needs to do can be accomplished by obtaining a pointer to an interface and calling its member functions. Interfaces, in fact, make up more of the so-called "Application Programming Interface" for a user of an object and define the implementation for an object itself instead of using more archaic mechanisms such as explicitly named exports, callback functions, and messages. There are only a few truly fundamental API functions in OLE 2, and most are concerned with creating objects or manipulating things such as class IDs.

The majority of the hundred or so OLE 2 API functions are actually "wrappers" for sequences of commonly used interface calls. Some are the equivalent of calling *QueryInterface* for a specific interface, calling a member function with default parameters, and releasing that interface (Call-Use-Release again). Such wrappers are provided to simplify application development in most cases; their use is seldom required, but you can benefit from the

convenience. Even then, typical compound document containers or objects, even with full in-place activation and drag-and-drop implementation, will generally use about 20 of these API functions. Some you might use for very specific reasons; others, although they exist, you probably will never use.

If you are familiar with the OLE 1 API, you'll find that many operations that were API functions in OLE 1, such as *OleSetHostNames*, are replaced by an interface call in OLE 2, such as *IOleObject::SetHostNames*. Many more should become apparent as we implement features using OLE 2.

A major advantage of defining new functions as interfaces is that people outside Microsoft can publish a new interface by providing an include file that defines the interface and possibly by providing a marshaling DLL if they want that interface to be implementable in an EXE. This means no update of the operating system is required to accommodate your functions, and you and others can immediately start using those interfaces without waiting for Microsoft to revise the system.

What Is a Windows Object? (Reprise)

A Windows Object is any object, in whatever form it manifests itself, that supports at least one interface, *IUnknown*. A Windows Object must be able to provide a separate function table for each interface it supports. The implementation of *IUnknown* members in each supported interface must be aware of the entire object because it must be able to access all other interfaces in the object and it must be able to affect the object's reference count.

C++ multiple inheritance is a convenient way to provide multiple function tables for each interface as the compiler generates them automatically. Because each implementation of a member function is already part of your object class, each automatically has access to everything in the object.

However, because this book is intended to help both C and C++ programmers, I will take a different approach. The object class itself will simply inherit from *IUnknown* and implement these functions to control the object as a whole. Each interface supported by this object is implemented in a separate C++ class that singly inherits from the interface it is implementing. These "interface implementations" are instantiated with the object and live as long as the object lives.

The *IUnknown* members of these interface implementations always delegate to some other *IUnknown* implementation, which in most cases is the overall object's *IUnknown*. Each interface implementation also holds a "back

pointer" to the object in which the implementations are contained so that they are able to access information centrally stored in the object. In C++, this generally requires that each interface implementation class be a *friend* of the object class. It is also highly useful to maintain an interface-level reference count for debugging.

You might still have one question: What about inheritance for Windows Objects? Can one Windows Object inherit from another? The truth is that there is no inheritance mechanism because inheritance is a way to achieve code that you can reuse. The Windows Object mechanism for reuse is called *aggregation*. But, alas, we are beginning to discuss the finer details of implementation. So with that, we can close this chapter.

Summary

An object in object-oriented terms is a self-contained unit of data and functions to manipulate that data. A Windows Object is a special manifestation of this definition that presents its functions as separate groups called interfaces. Windows Objects differ from C++ objects in construction and use, but they are more powerful than C++ objects because they can live anywhere on the system and still be as usable to an application as if they were incorporated into that application. The most fundamental question that forms a theme for this book is how to obtain the first interface pointer for a variety of objects and what you can do with that pointer after you obtain it.

The most basic functions of all interface pointers are concerned with reference counting and with obtaining other interface pointers to the same object. These functions are collected in an OLE 2 interface named *IUnknown*. Later chapters in this book deal with more specific types of objects, their special interfaces, how you obtain those interface pointers, and what you can do with them.

Implementations and users of objects written in C and C++ differ only slightly. Calling a member function by means of a C pointer requires an extra dereference through a pointer to an interface function table and passage of an extra parameter to simulate C++'s *this* pointer. C objects must manually construct the function tables for their interfaces. Although C++ is more convenient, it is not the required language of OLE 2.

A type of object called an enumerator provides functions through *IEnum* interfaces to iterate over a list of elements. Because Windows Objects are portable across process boundaries, an enumerator object is used to pass lists of information across those same boundaries as well as to pass the functions to iterate over that list.

OLE 2 interface members use a specific *cdecl* calling convention, and the OLE 2 header files define a number of macros to isolate machine specifics from the definitions of interfaces. Most interface members also return a type called an HRESULT, which contains detailed error information.

OLE 2 defines a number of standard interfaces but allows those objects to be implemented in DLLs to define and implement custom interfaces. If you provide custom marshaling, a topic not covered in this book, you can also provide custom interfaces from objects implemented in EXEs as well. Although OLE 2 defines many interfaces, applications need only worry about a handful, depending on the features those applications want to implement.

CHAPTER FOUR

COMPONENT OBJECTS (THE COMPONENT OBJECT MODEL)

Arthur: Camelot!
Galahad: Camelot...
Launcelot: Camelot...
Patsy: It's only a model....
Arthur: Sh!

From *Monty Python and the Holy Grail*

Almost everyone who has tried to present all the material in OLE 2 to an audience in a comprehensible way has tried to portray "The Component Object Model" as a "feature." I'm certainly guilty of trying this once, but when I did, I had never seen so many glazed looks in my life. Those who didn't have glazed expressions seemed to be saying "So what? What can you do with a model?" To clear the air and to redeem myself somewhat, this chapter is about using and implementing very general Windows Objects that involve the fundamental API functions and interfaces *specified* in the Component Object Model and *implemented* in COMPOBJ.DLL, referred to in this book as the Component Object library.

The Component Object Model—the specification—is mostly about interfaces, reference counting, and *QueryInterface* (that is, the basic standardization of a Windows Object). The Component Object library—the implementation—is a number of fundamental API functions, also specified in the model, that provide for object creation and management, as well as code that handles marshaling of interface function calls across process boundaries. This implementation provides one answer to the ultimate question posed in

Chapter 3 in the form of a "component object." A component object is a Windows Object identified by a unique class identifier (CLSID) that associates an object with a particular DLL or EXE in your file system. To obtain a pointer to a component object, you pass a CLSID to one of two Component Object library API functions. The library in turn locates and loads the code implementing that object, instantiates the object, and asks the object for an interface pointer to return to you. Note that compound document objects are merely special cases of the more general component object, so the discussion here is relevant if you are interested in implementing a compound document object server.

Before diving into the subject of using and implementing component objects, we must first discuss a few requirements of all applications (EXEs) that either use or implement objects. Applications (which define a task) must initialize the Component Object library before using any other OLE 2 API functions (from any OLE 2 DLL), and part of this initialization has to do with memory management within the application's task. Because both operations are crucial and are used in all remaining sample applications in this book, initialization and memory management will be the first two topics of this chapter.

The code using a component object, which we can refer to as a *component user,* calls the library only to obtain that first pointer to an object identified by a CLSID. The overall impact on such a component user is minimal, as this chapter will demonstrate. The component user need not be concerned about where the code for the object is actually located or how the object is implemented. The greater impact is on the implementation of a component object that allows the library to locate, load, and instantiate it, based on a CLSID. To accommodate such a capability, you must implement a standard structure around the object—one structure for EXEs, another structure for DLLs. In addition, you must store information in the registration database under your object's CLSID, which identifies the name of your object and where it lives. The Component Object library uses this information to find your object and connect it to its component user.

This chapter will demonstrate a simple object implemented in both a DLL and an EXE, as well as a user of those objects. We will also implement Cosmo's Polyline object as a component object in a DLL that we'll carry forward through subsequent chapters as we add more OLE 2 features.

This chapter closes with a discussion about Windows Object reusability through a mechanism called *aggregation*. One object, called the *aggregate*, internally creates instances of other objects, possibly exposing the interfaces of those objects as an interface on the aggregate. Aggregation accomplishes code reuse as C++ inheritance does but without the problems of inheritance. Although the topic is appropriate to discuss here, we won't actually put it to use until later chapters. You might, however, find it a useful mechanism around which to design reuse of code in your own applications.

So, to explain what the Component Object Model is, we really need to analyze the model's impact on applications in general, on the user of a component object, and on the implementation of a component object. The specifications of the Component Object Model provide the foundation for how Windows will evolve from an API-based system to an object-oriented system, a romantic walk through the lush gardens of Camelot. But that could be another whole topic in itself, so we'll stick to implementation details; it is, after all, only a model.

Where the Wild Things Are (with Apologies to Maurice Sendak)

There are component users and component objects, both of which can reside in any piece of code, EXE and DLL alike. The system features provided in OLE 2 are themselves both objects and users, all of which live in DLLs. The implementation portion of the Component Object library is considered part of the OLE 2 system features.

Whether an object user lives in an EXE or a DLL is of little importance—EXEs have a little more work, as described in the next section, because they define a task. In any case, regardless of where an object user lives, we can illustrate the relationship between object and user, as shown in Figure 4-1 on the next page. Note that the word "server" in this figure applies to the module that services an object, either an EXE or a DLL.

Because an EXE object requires marshaling support, performance is typically slower than with a DLL object. But there is one major benefit to having this marshaling support: A 16-bit DLL (and any objects it implements) cannot be loaded into a 32-bit process space of the object user, nor can a 32-bit DLL be loaded into a 16-bit process space. The marshaling code in COMPOBJ.DLL, however, knows how to pass parameters between 16-bit and 32-bit processes, thereby allowing an object user in one space to communicate with an object in another space.

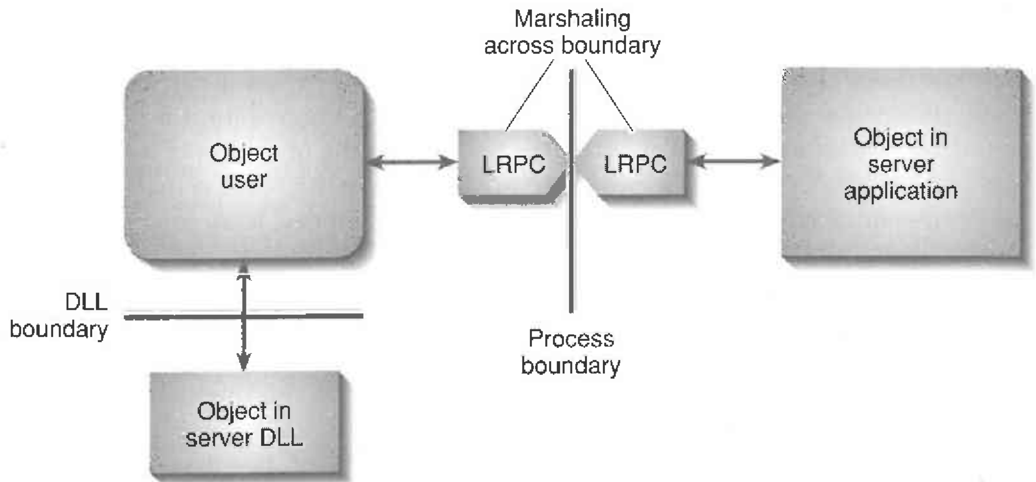


Figure 4-1.
Component users (in DLLs or EXEs) see other component objects either in DLLs or in other EXEs. The Component Object library lives between the user and an EXE object to provide marshaling. There is no mediator between the user and a DLL object.

The Component Object library is the agent responsible for getting at the first object interface pointer in any series of communications between object and user. It worries about making sure the right piece of object code is in memory whenever something else wants to use that object. However, once the initial object has been created and is handed to the user, the object and the user can create other objects themselves and pass them to their partners. The Component Object library is only there for marshaling and memory management (if even necessary) and is otherwise out of the picture. The objects we implement in this chapter require a certain structure and registration such that any user can instantiate an object through the Component Object library API. Because this API is used underneath much of the Compound Document API (those functions prefixed with *Ole*, as in *OleCreate*), all compound document objects are also objects that fit this model—they simply have a more precisely defined behavior. However, the Component Object library is highly useful for creating component software, and Windows itself is headed in this direction: not to be a Compound Document system, but to be a Component Object system.

Compound Document Terminology

A number of the terms generally used in discussions of compound documents are used in this chapter to discuss much more generic concepts. The following list provides the crucial compound document terms and describes how they apply to the information in this chapter:

Container A user of compound document objects. A container exposes site objects to the compound document objects they contain, but those site objects are not separately addressable components and are passed to the contained object only at run-time. However, site objects are Windows Objects.

Server (sometimes just object) An implementor of an object, either a DLL or an EXE. A server may implement a component object or a compound document object and expose both through identical structures usable by the Component Object library.

In-Process Server or DLL Server A server of objects specifically implemented in a DLL.

Server Application or EXE Server A server of objects specifically implemented in an EXE. Sometimes called an *object application*.

Object Handler A lightweight DLL server containing a partial implementation of an object that is fully implemented elsewhere in an EXE. Handlers are not expected to implement complete objects (especially not editing capabilities) and are intended for redistribution. Structurally they are identical to DLL servers.

The New Application for Windows Objects

Any and all applications that plan to *use or implement* Windows Objects (not just component objects) must ensure that the Component Object library is properly initialized before attempting to use other OLE 2 API functions. In addition, applications running under Windows 3.1 that intend to use objects implemented in other applications must make special considerations for Lightweight Remote Procedure Call (LRPC) use of *PostMessage*. An object or

an object user in a DLL need not be concerned with any of these requirements that apply only to the application that defines a task. For applications (EXEs), here are the steps for initialization:

1. Call the Windows API *SetMessageQueue(96)* to set your application's message queue size to 96, if possible. This is the recommended size for LRPC handling. This function is not necessary in Win32 because Win32 message queues size dynamically.
2. Verify the library build version by calling *CoBuildVersion* or *OleBuildVersion*.
3. Call *CoInitialize* or *OleInitialize* on startup.
4. Call *CoUninitialze* or *OleUninitialize* when shutting down to allow the DLL objects to be freed if and only if step 3 worked.

NOTE: *CoBuildVersion, CoInitialize,* and *CoUninitialize* have counterparts with *Ole* prefixes: *OleBuildVersion, OleInitialize,* and *OleUnintialize.* The *Co...* functions control your access to Component Object library functions. If you use any clipboard, drag-and-drop, Compound Document, or Automation related API functions, you must use the *Ole...* functions instead of their *Co...* counterparts. The *Ole...* versions simply perform a few more specific operations and call the *Co...* versions. Compound Document applications, containers included, *always* use the *Ole...* versions.

Absolutely all of the sample applications in this book that compile EXEs include these four steps. Most of the samples in this chapter, as well as those in Chapters 5 and 6, use the *Co...* variants. All samples in Chapter 7 and beyond will use the *Ole...* functions because those samples will in turn depend on the "extras" provided by the *Ole...* functions. The first sample in the later section "Memory Management and Allocator Objects" will demonstrate each of these steps. In the meantime, let's look at each step in detail and examine why each is necessary.

Enlarge the Message Queue

OLE 2's LRPC implementation works on top of the Windows API function *PostMessage.* In a nutshell, when the user of an object in another application calls one of the object's member functions, the function generates an LRPC call, which in actuality is a *PostMessage* from the one application process space into the other. To handle all the possible *PostMessage* traffic, Microsoft

recommends that all OLE 2 applications with even the slightest chance of engaging in LRPC calls call *SetMessageQueue* set to *96* on startup, if possible. Something like the following should, in fact, be your very first step inside *WinMain* to ensure that no messages yet exist in your queue because *SetMessageQueue* will destroy anything already there:

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    [variables, but NO code]
    int      cMsg=96;

   #ifndef WIN32
    //Enlarge the queue as large as we can starting from 96
    while (!SetMessageQueue(cMsg) && (cMsg-=8));
   #endif

    [Initialization code, message loop, etc.]
    }
```

If you don't enlarge your message queue sufficiently, the Component Object library could reject some LRPC calls when your queue is full. Enlarging your message queue provides sufficient space for LRPC traffic.

Verify the Library Build Version

Before using any other Component Object API (*Co...*) function, an application should call *CoBuildVersion(void)* to get major and minor build numbers in a returned DWORD. If you are planning to go on to use OLE 2's data transfer, Compound Document, or Automation technologies, you must instead call *OleBuildVersion(void)*, which returns a similar DWORD. The high-order word of the return value is a major version number, and the low-order word is the minor version number.

An application can run against only one major version of the libraries, but it can run against any minor version. The version numbers you can run against are compiled into your application as the symbols *rmm* (major) and *rup* (minor) defined in OLE2VER.H. (There is also a *rmj* symbol, which might look like the "major" number but is unfortunately not used this way.) Note that these numbers are not product release numbers—that is, in OLE 2 these are not *2* and *0*. Do not depend on any interpretation of these numbers. With these numbers, you must compare your *rmm* to the major version of the libraries, and if they do not match, you must fail loading your application as shown on the next page.

```
#include <compobj.h>     //For Ole... functions, use OLE2.H
#include <ole2ver.h>

...

DWORD    dwVer;

dwVer=CoBuildVersion();    //Or OleBuildVersion

if (rmm==HIWORD(dwVer))
    {
    //Major versions match.

    if (rup <= LOWORD(dwVer))
        {
        //Library is newer than or as old as the app; use normally.
        }
    else
        {
        /*
         * App was written for newer libraries. Disable features
         * that depend on API or bug fixes in newer libraries
         * or simply fail altogether.
         */
        }
    }
else
    //Major version mismatch; fail loading application.
```

Minor version numbers are useful to applications that want to know whether the libraries they've loaded contain a particular function or have a specific bug fix. Let's say minor version 12 of OLE 2 added a function that improves performance over minor version 11. If I load the minor version 11 libraries, I cannot attempt to call that version 12 function. If, however, I find that I am running against minor version 12, I can take advantage of what's available.

Call *CoInitialize* or *OleInitialize*

On startup, an application must call *CoInitialize* or *OleInitialize* before calling any other function in either of their respective libraries. You must use *OleInitialize* for any feature other than component objects and compound files, including all data transfer, drag-and-drop, compound documents, and even Automation. Component Object library and Compound File API functions can be used after only *CoInitialize*:

```
if (FAILED(CoInitialize(NULL)))    //Or OleInitialize
    [Fail loading the application].

m_fInitialized=TRUE;
```

Both functions identically take a pointer to an *allocator object* that supports the *IMalloc* interface. Through this object, all other parts of this application and the DLLs that live in this application's task can allocate task local memory (as opposed to shared memory). If NULL is passed, as shown in the preceding example, OLE uses a default allocator in COMPOBJ.DLL. Any code in this application or in a DLL loaded into this task can call the *CoGetMalloc* API function to retrieve an *IMalloc* pointer to this same allocator. We'll see this in more detail in the section "Memory Management and Allocator Objects."

Any code within the same task can call *CoInitialize* multiple times; in such circumstances, the *IMalloc* passed to the first *CoInitialize* wins. This allows any code (usually that in a DLL) to call *CoInitialize* to ensure that it can use the Component Object library even if the application that loaded the DLL did not make the call.[1]

When *CoInitialize* is called more than once in the same task, it will return an HRESULT with S_FALSE—a code that does not mean failure but that means nothing happened. As we have seen, the FAILED() macro will return FALSE for S_FALSE just as it will for S_OK, so the preceding code fragment is valid for all uses of *CoInitialize*.

An application must remember whether *CoInitialize* or *OleInitialize* worked (in a variable such as *m_fInitialized*) so that it knows whether to call *CoUninitialize* or *OleUninitialize* when it shuts down. In other words, every ...*Uninitialize* call must be matched one-to-one with an ...*Initialize* call.

Call *CoUninitialize* or *OleUninitialize*

After an application has finished with the libraries, it must call *CoUninitialize* if it previously called *CoInitialize*, or it must call *OleUninitialize* if it previously called *OleInitialize*. Neither function takes any parameters. You should remember whether the ...*Initialize* call succeeded and call ...*Uninitialize* only if it did—that is, balance the calls as you would balance *GlobalAlloc* and *GlobalFree*.

---

1. Microsoft recommends that DLLs always pass NULL to *CoInitialize*. Note also that because your DLL's *LibMain* is called before the application's *WinMain*, *CoInitialize* will never have been called by that time. It's best to defer any dependencies on the task allocator until after *LibMain*, if possible.

Internally, *OleUninitialize* cleans up the specifics from *OleInitialize* and calls *CoUninitialize*. This latter function will call another *CoFreeAllLibraries*, which forcibly and unconditionally unloads all object DLLs that were loaded on behalf of the application. That's why you must be careful when you call *CoUninitialize*. You might have use for *CoFreeAllLibraries* yourself if your application's debugging version has the ability to suddenly terminate and unload (say, on an assert failure), which might not normally call *CoUninitialize*.

Memory Management and Allocator Objects

Up to now, the only system-supported memory management functions have been the various *Local*... and *Global*... Windows API functions (*LocalAlloc*, *LocalFree*, *GlobalAlloc*, *GlobalFree*, and so on). OLE 2 introduces a new object-oriented technique to deal with memory management through the use of *allocator objects*. Within any given task—that is, within a process space in which a single EXE is running—there is a single task allocator object and a single shared allocator object. The application can implement the task allocator, or it can use the default task allocator implemented in the Component Object library. The shared allocator is not replaceable—the implementation in the Component Object library is always used to ensure that memory is truly shareable.

You specify the task allocator through the only parameter to *CoInitialize* or *OleInitialize*. This parameter is an *IMalloc* pointer to whatever allocator object defined memory management in the task. A NULL pointer means "use the default task allocator," whereas a non-NULL pointer means "use this application-implemented task allocator."

An allocator object implements the *IMalloc* interface, defined in COMPOBJ.H. The *IMalloc* interface describes most of the same functions that Windows provides for local and global memory, such as *LocalAlloc*, *LocalFree*, and *LocalCompact*. For specific details on each member function, see the *OLE 2 Programmer's Reference*, but it's fairly easy to guess at how to use each function in this interface based on their signatures alone. (*IUnknown* members have been removed from the following listing for brevity; you will see them explicitly in the include file.)

```
DECLARE_INTERFACE_(IMalloc, IUnknown)
    {
    STDMETHOD_(void FAR*, Alloc) (THIS_ ULONG cb) PURE;
    STDMETHOD_(void FAR*, Realloc) (THIS_ void FAR * pv, ULONG cb) PURE;
```

```
    STDMETHOD_(void, Free) (THIS_ void FAR * pv) PURE;
    STDMETHOD_(ULONG, GetSize) (THIS_ void FAR * pv) PURE;
    STDMETHOD_(int, DidAlloc) (THIS_ void FAR * pv) PURE;
    STDMETHOD_(void, HeapMinimize) (THIS) PURE;
    };

typedef        IMalloc FAR * LPMALLOC;
```

At any time, any piece of code in the application or any DLL loaded into this task (including the OLE 2 libraries) can and will call *CoGetMalloc* to obtain an *IMalloc* pointer to the task allocator object. In other words, a task allocator is a Windows Object, and you use the API function *CoGetMalloc* to obtain the first interface pointer. In this case, your application might be the object implementor and OLE 2 might be the object user. It works both ways.

All the OLE 2 libraries always use the task allocator for all non-shared memory needs. Some OLE 2 functions will allocate memory using the task allocator then pass a pointer to that memory to your application where you become responsible to free it. In such cases you must free the memory when you no longer need it by calling *CoGetMalloc* to obtain the task allocator's *IMalloc* pointer, pass the memory pointer to *IMalloc::Free*, and finish up by calling *IMalloc::Release*.

The first parameter to *CoGetMalloc* is either MEMCTX_TASK or MEMCTX_SHARED, depending on which allocator object you want. The second parameter is a pointer to an LPMALLOC variable that receives the *IMalloc* pointer.

The default task allocator is based on multiple local heap management (or far local heaps). This allocator allows you to allocate more than 64 KB because there are multiple heaps, but each allocation is as efficient as a local allocation because you use only one selector per heap instead of one per allocation (as *GlobalAlloc* does). The only limitation is that any single allocation must be smaller than 64 KB. The shared allocator provides memory that different processes can independently access and is built on the same type of multiple local heap management as the default task allocator. Likewise, all shared allocations must be 64 KB or smaller.

The *Malloc* program (CHAP04\MALLOC), shown in Listing 4-1, exercises the functions in both the standard task allocator (it does not implement its own allocator) and the shared allocator. It is an application that is most interesting in a debugger, and it does indicate success or failure of its operations in message boxes. (Yes, a most advanced user interface.)

MALLOC.H

```
/*
 * IMalloc Demonstration Chapter 4
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _MALLOC_H_
#define _MALLOC_H_

#include <BOOK1632.H>

//Menu Resource ID and Commands
#define IDR_MENU                            1

#define IDM_IMALLOCCOGETMALLOCTASK          100
#define IDM_IMALLOCCOGETMALLOCSHARED        101
#define IDM_IMALLOCRELEASE                  102
#define IDM_IMALLOCALLOC                    103
#define IDM_IMALLOCFREE                     104
#define IDM_IMALLOCREALLOC                  105
#define IDM_IMALLOCGETSIZE                  106
#define IDM_IMALLOCDIDALLOC                 107
#define IDM_IMALLOCHEAPMINIMIZE             108
#define IDM_IMALLOCEXIT                     109

//MALLOC.CPP
LRESULT FAR PASCAL EXPORT MallocWndProc(HWND, UINT, WPARAM
    , LPARAM);

#define CALLOCS 10

/*
 * Application-defined classes and types.
 */

class __far CAppVars
    {
    friend LRESULT FAR PASCAL EXPORT MallocWndProc(HWND, UINT
        , WPARAM, LPARAM);

    protected:
        HINSTANCE  m_hInst;           //WinMain parameters
        HINSTANCE  m_hInstPrev;
        UINT       m_nCmdShow;
```

(continued)

Listing 4-1.
The IMalloc program, which exercises task and shared allocator objects.

Listing 4-1. *continued*

```
        HWND        m_hWnd;             //Main window handle
        LPMALLOC    m_pIMalloc;         //IMalloc interface
        BOOL        m_fInitialized;     //Did CoInitialize work?

        ULONG       m_rgcb[CALLOCS];    //Sizes to allocate
        LPVOID      m_rgpv[CALLOCS];    //Allocated pointers

    public:
        CAppVars(HINSTANCE, HINSTANCE, UINT);
        ~CAppVars(void);
        BOOL FInit(void);

        void FreeAllocations(BOOL);
    };

typedef CAppVars FAR *LPAPPVARS;

#define CBWNDEXTRA              sizeof(LONG)
#define MALLOCWL_STRUCTURE      0

#endif //_MALLOC_H_
```

MALLOC.CPP

```
/*
 * IMalloc Demonstration Chapter 4
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include <windows.h>
#include <ole2.h>
#include <initguid.h>
#include <ole2ver.h>
#include "malloc.h"

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    MSG         msg;
    LPAPPVARS   pAV;
    int         cMsg=96;
```

(continued)

Listing 4-1. *continued*

```
    #ifndef WIN32
     while (!SetMessageQueue(cMsg) && (cMsg-=8));
    #endif

     pAV=new CAppVars(hInst, hInstPrev, nCmdShow);

     if (NULL==pAV)
         return -1;

     if (pAV->FInit())
         {
         while (GetMessage(&msg, NULL, 0,0 ))
             {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
             }
         }

     delete pAV;
     return msg.wParam;
     }

LRESULT FAR PASCAL EXPORT MallocWndProc(HWND hWnd, UINT iMsg
     , WPARAM wParam, LPARAM lParam)
     {
     LPAPPVARS       pAV;
     LPVOID          pv;
     ULONG           cb;
     UINT            i;
     BOOL            fResult=TRUE;
     HRESULT         hr;

     pAV=(LPAPPVARS)GetWindowLong(hWnd, MALLOCWL_STRUCTURE);

     switch (iMsg)
         {
         case WM_NCCREATE:
             pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)
             ->lpCreateParams);

             SetWindowLong(hWnd, MALLOCWL_STRUCTURE, (LONG)pAV);
             return (DefWindowProc(hWnd, iMsg, wParam, lParam));

         case WM_DESTROY:
             PostQuitMessage(0);
             break;
```

(continued)

Listing 4-1. *continued*

```
        case WM_COMMAND:
            switch (LOWORD(wParam))
                {
                case IDM_IMALLOCCOGETMALLOCTASK:
                    pAV->FreeAllocations(TRUE);

                    hr=CoGetMalloc(MEMCTX_TASK, &pAV->m_pIMalloc);
                    fResult=SUCCEEDED(hr);

                    MessageBox(hWnd, ((fResult)
                        ? "CoGetMalloc(task) succeeded."
                        : "CoGetMalloc(task) failed.")
                        , "Malloc", MB_OK);

                    break;

                case IDM_IMALLOCCOGETMALLOCSHARED:
                    pAV->FreeAllocations(TRUE);
                    hr=CoGetMalloc(MEMCTX_SHARED, &pAV->m_pIMalloc);
                    fResult=SUCCEEDED(hr);

                    MessageBox(hWnd, ((fResult)
                        ? "CoGetMalloc(shared) succeeded."
                        : "CoGetMalloc(shared) failed.")
                        , "Malloc", MB_OK);

                    break;

                case IDM_IMALLOCRELEASE:
                    pAV->FreeAllocations(TRUE);
                    break;

                case IDM_IMALLOCALLOC:
                    if (NULL==pAV->m_pIMalloc)
                        break;

                    pAV->FreeAllocations(FALSE);

                    for (i=0; i < CALLOCS; i++)
                        {
                        LPBYTE      pb;
                        ULONG       iByte;

                        cb=pAV->m_rgcb[i];
                        pAV->m_rgpv[i]=pAV->m_pIMalloc->Alloc(cb);
```

(continued)

Listing 4-1. *continued*

```
                //Fill the memory with letters.
                pb=(LPBYTE)pAV->m_rgpv[i];

                if (NULL!=pb)
                    {
                    for (iByte=0; iByte < cb; iByte++)
                        *pb++=('a'+i);
                    }

                fResult &= (NULL!=pAV->m_rgpv[i]);
                }

            MessageBox(hWnd, ((fResult)
                    ? "IMalloc::Alloc succeeded."
                    : "IMalloc::Alloc failed.")
                    , "Malloc", MB_OK);

            break;

        case IDM_IMALLOCFREE:
            pAV->FreeAllocations(FALSE);

            MessageBox(hWnd, "IMalloc::Free finished."
                , "Malloc", MB_OK);
            break;

        case IDM_IMALLOCREALLOC:
            if (NULL==pAV->m_pIMalloc)
                break;

            for (i=0; i < CALLOCS; i++)
                {
                LPBYTE      pb;
                ULONG       iByte;

                pAV->m_rgcb[i]+=128;

                //Old memory is not freed if Realloc fails.
                pv=pAV->m_pIMalloc->Realloc(pAV->m_rgpv[i]
                    , pAV->m_rgcb[i]);

                if (NULL!=pv)
                    {
```

(continued)

Listing 4-1. *continued*

```
            pAV->m_rgpv[i]=pv;

            //Fill the new memory
            //with something we can see.
            pb=(LPBYTE)pAV->m_rgpv[i];
            cb=pAV->m_rgcb[i];

            if (NULL!=pb)
                {
                for (iByte=cb-128; iByte
                    < cb; iByte++)
                    {
                    *pb++=('a'+i);
                    }
                }
            }
        else
            fResult=FALSE;
        }

    MessageBox(hWnd, ((fResult)
        ? "IMalloc::Realloc succeeded."
        : "IMalloc::Realloc failed.")
        , "Malloc", MB_OK);

    break;

case IDM_IMALLOCGETSIZE:
    if (NULL==pAV->m_pIMalloc)
        break;

    for (i=0; i < CALLOCS; i++)
        {
        cb=pAV->m_pIMalloc->GetSize(pAV->m_rgpv[i]);

        /*
         * We test that the size is *at least*
         * what we wanted.
        /*
        fResult &= (pAV->m_rgcb[i] <= cb);
        }

    MessageBox(hWnd, ((fResult)
        ? "IMalloc::GetSize matched."
```

(continued)

Listing 4-1. *continued*

```
                    : "IMalloc::GetSize mismatch."), "Malloc"
                    , MB_OK);

                break;

            case IDM_IMALLOCDIDALLOC:
                if (NULL==pAV->m_pIMalloc)
                    break;

                /*
                 * DidAlloc may return -1 if it does not know if
                 * it actually allocated something. In that
                 * case, we just blindly & in a -1 with no effect.
                 */
                for (i=0; i < CALLOCS; i++)
                    {
                    fResult &= pAV->m_pIMalloc->DidAlloc(pAV
                        -> m_rgpv[i]);
                    }

                MessageBox(hWnd, ((fResult)
                    ? "IMalloc::DidAlloc is TRUE."
                    : "IMalloc::DidAlloc is FALSE.")
                    , "Malloc", MB_OK);

                break;

            case IDM_IMALLOCHEAPMINIMIZE:
                if (NULL!=pAV->m_pIMalloc)
                    pAV->m_pIMalloc->HeapMinimize();

                MessageBox(hWnd
                    , "IMalloc::HeapMinimize finished."
                    , "Malloc", MB_OK);

                break;

            case IDM_IMALLOCEXIT:
                PostMessage(hWnd, WM_CLOSE, 0, 0L);
                break;
            }
        break;

    default:
        return (DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
```

(continued)

Listing 4-1. *continued*

```
    return 0L;
    }

CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev
    , UINT nCmdShow)
    {
    UINT        i;
    ULONG       cb;

    m_hInst       =hInst;
    m_hInstPrev   =hInstPrev;
    m_nCmdShow    =nCmdShow;

    m_hWnd        =NULL;
    m_pIMalloc    =NULL;
    m_fInitialized=FALSE;

    //100 is arbitrary.  IMalloc can handle larger.
    cb=100;

    for (i=0; i < CALLOCS; i++)
        {
        m_rgcb[i]=cb;
        m_rgpv[i]=NULL;
        cb*=2;
        }

    return;
    }

CAppVars::~CAppVars(void)
    {
    FreeAllocations(TRUE);

    if (m_fInitialized)
        CoUninitialize();

    return;
    }

BOOL CAppVars::FInit(void)
    {
    WNDCLASS    wc;
    DWORD       dwVer;

    //Make sure COMPOBJ.DLL is the right version
    dwVer=CoBuildVersion();
```

(continued)

Listing 4-1. *continued*

```
    if (rmm!=HIWORD(dwVer))
        return FALSE;

    //Call CoInitialize so that we can call other Co... functions
    if (FAILED(CoInitialize(NULL)))
        return FALSE;

    m_fInitialized=TRUE;

    if (!m_hInstPrev)
        {
        wc.style            = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc      = MallocWndProc;
        wc.cbClsExtra       = 0;
        wc.cbWndExtra       = CBWNDEXTRA;
        wc.hInstance        = m_hInst;
        wc.hIcon            = LoadIcon(m_hInst, "Icon");
        wc.hCursor          = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground    = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName     = MAKEINTRESOURCE(IDR_MENU);
        wc.lpszClassName    = "MALLOC";

        if (!RegisterClass(&wc))
            return FALSE;
        }

    m_hWnd=CreateWindow("MALLOC", "IMalloc Object Demo"
        , WS_OVERLAPPEDWINDOW, 35, 35, 350, 250, NULL, NULL
        , m_hInst, this);

    if (NULL==m_hWnd)
        return FALSE;

    ShowWindow(m_hWnd, m_nCmdShow);
    UpdateWindow(m_hWnd);

    return TRUE;
    }

void CAppVars::FreeAllocations(BOOL fRelease)
    {
    UINT    i;

    if (NULL==m_pIMalloc)
        return;
```

(continued)

Listing 4-1. *continued*

```
    for (i=0; i < CALLOCS; i++)
        {
        if (NULL!=m_rgpv[i])
            m_pIMalloc->Free(m_rgpv[i]);

        m_rgpv[i]=NULL;
        }

    if (fRelease)
        {
        m_pIMalloc->Release();
        m_pIMalloc=NULL;
        }

    return;
    }
```

Using the Heapwalker application in the Windows SDK, we can see where OLE 2 allocates each type of memory—task and shared—using its own allocator objects. As shown in Figure 4-2, task memory is allocated from multiple heaps belonging to the MALLOC application task.

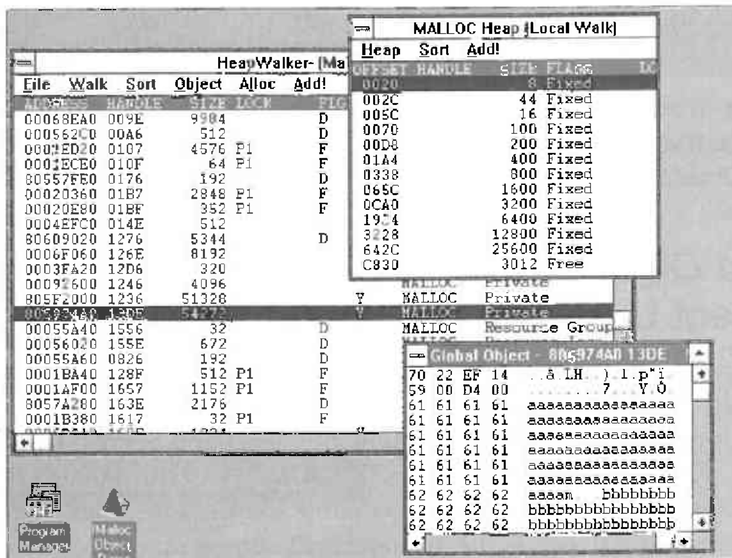


Figure 4-2.
Using Local Walk on the heap shows allocated blocks. The blocks are filled with letters to show their location in a hexadecimal dump.

When this memory is freed, the heaps are not necessarily freed, but the space inside those heaps are freed, as shown in Figure 4-3.

Shared memory is allocated on behalf of COMPOBJ.DLL, as shown in Figure 4-4, again using the same heap management technique that the standard task allocator uses. Freeing memory generates the same results for the task allocator, as shown in Figure 4-3.

Figure 4-3.
Using Local Walk on the heap, after freeing the memory, shows free space in the heaps. Freed blocks are filled with 0xCC.

Component Objects from Class Identifiers: A Component User

Let's suppose I'm an application and I know there exists a Windows Object named Koala (which is, in fact, the name of an object we'll implement in the later section "Implementing a Component Object and Server"). I can identify the Koala object using the CLSID "00021102-0000-0000-C000-000000000046." (Whoa! That's a long name there! Remember that CLSIDs are 128 bits: This string is the hexadecimal representation of those bits.) Koalas are nice; they come with an include file that defines a CLSID_Koala constant. Let's say I also know that the Koala object supports the interface

Figure 4-4.
Using Local Walk on the heap shows allocated blocks exactly as the task allocator does, but owned by COMPOBJ.DLL instead of by the application.

IPersist, which is a very simple interface capable only of returning the CLSID of its object. Given this knowledge, how do I create a Koala object with this CLSID and obtain a pointer to its *IPersist* interface?

This question should ring a harmonic with the ultimate question posed in Chapter 3, so let's look at the answer. For the benefit of those readers who will be writing compound object container applications, I want to mention that the API functions and interface functions that we use to instantiate a component object are used within more complex API functions that we'll use in Chapter 9 to instantiate a compound document object. Again, compound document objects are more refined and specialized component objects; what we discuss here is simply the logical equivalent of calling the C++ *new* operator. If you are in a hurry to implement a compound document container, you can skip to the next chapter after you finish reading this section.

The OBJUSER program, in Listing 4-2, implements a component user of the Koala component objects that we'll implement in the next section. Koala implements the *IPersist* interface only, but by virtue of implementing one interface, it also implements *IUnknown*: *IPersist* includes all *IUnknown* member functions plus one other named *GetClassID,* which returns (what else?) the CLSID of the object. *IPersist* is a base interface in OLE 2 for a

number of other interfaces, and rarely is it used by itself. I chose *IPersist* for this demonstration because it has standard marshaling support already in COMPOBJ.DLL. This built-in support means we can implement marshaling in both DLLs and EXEs without any extra work. I also did not use another OLE 2 interface because most other interfaces have more member functions we would have to implement and would have raised more questions than we're prepared to deal with now.

OBJUSER.H

```
/*
 * Koala Object User Chapter 4
 *
 * Definitions and structures.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _OBJUSER_H_
#define _OBJUSER_H_

#include <bookguid.h>

//Menu Resource ID and Commands
#define IDR_MENU                        1

#define IDM_OBJECTUSEDLL                100
#define IDM_OBJECTUSEEXE                101
#define IDM_OBJECTCREATECOGCO           102
#define IDM_OBJECTCREATECOCI            103
#define IDM_OBJECTRELEASE               104
#define IDM_OBJECTGETCLASSID            105
#define IDM_OBJECTEXIT                  106

//OBJUSER.CPP
LRESULT FAR PASCAL EXPORT ObjectUserWndProc(HWND, UINT, WPARAM
    , LPARAM);

class __far CAppVars
    {
    friend LRESULT FAR PASCAL EXPORT ObjectUserWndProc(HWND
        , UINT, WPARAM, LPARAM);
```

Listing 4-2. *(continued)*

The OBJUSER program, which uses Koala objects.

Listing 4-2. *continued*

```
    protected:
        HINSTANCE  m_hInst;          //WinMain parameters
        HINSTANCE  m_hInstPrev;
        UINT       m_nCmdShow;

        HWND       m_hWnd;           //Main window handle
        BOOL       m_fEXE;           //Menu selection

        LPPERSIST  m_pIPersist;      //IPersist interface
        BOOL       m_fInitialized;   //Did CoInitialize work?

    public:
        CAppVars(HINSTANCE, HINSTANCE, UINT);
        ~CAppVars(void);
        BOOL FInit(void);
    };

typedef CAppVars FAR *LPAPPVARS;

#define CBWNDEXTRA               sizeof(LONG)
#define OBJUSERWL_STRUCTURE      0

#endif //_OBJUSER_H_
```

OBJUSER.CPP

```
/*
 * Koala Object User Chapter 4
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#define INITGUIDS
#include <windows.h>
#include <ole2.h>
#include <ole2ver.h>
#include "objuser.h"

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    MSG         msg;
    LPAPPVARS   pAV;
```

(continued)

Listing 4-2. *continued*

```
    int      cMsg=96;

   #ifndef WIN32
    //Enlarge the queue as large as we can starting from 96
    while (!SetMessageQueue(cMsg) && (cMsg-=8));
   #endif

    pAV=new CAppVars(hInst, hInstPrev, nCmdShow);

    if (NULL==pAV)
        return -1;

    if (pAV->FInit())
        {
        while (GetMessage(&msg, NULL, 0,0 ))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    delete pAV;
    return msg.wParam;
    }

LRESULT FAR PASCAL EXPORT ObjectUserWndProc(HWND hWnd, UINT iMsg
    , WPARAM wParam, LPARAM lParam)
    {
    HRESULT         hr;
    LPAPPVARS       pAV;
    CLSID           clsID;
    LPCLASSFACTORY  pIClassFactory;
    DWORD           dwClsCtx;

    pAV=(LPAPPVARS)GetWindowLong(hWnd, OBJUSERWL_STRUCTURE);

    switch (iMsg)
        {
        case WM_NCCREATE:
            pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)
                ->lpCreateParams);

            SetWindowLong(hWnd, OBJUSERWL_STRUCTURE, (LONG)pAV);
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
```

(continued)

Listing 4-2. *continued*

```
        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
                {
                case IDM_OBJECTUSEDLL:
                    pAV->m_fEXE=FALSE;
                    CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEDLL
                        , MF_CHECKED);
                    CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEEXE
                        , MF_UNCHECKED);
                    break;

                case IDM_OBJECTUSEEXE:
                    pAV->m_fEXE=TRUE;
                    CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEDLL
                        , MF_UNCHECKED);
                    CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEEXE
                        , MF_CHECKED);
                    break;

                case IDM_OBJECTCREATECOGCO:
                    if (NULL!=pAV->m_pIPersist)
                        {
                        pAV->m_pIPersist->Release();
                        pAV->m_pIPersist=NULL;
                        CoFreeUnusedLibraries();
                        }

                    dwClsCtx=(pAV->m_fEXE) ? CLSCTX_LOCAL_SERVER
                         : CLSCTX_INPROC_SERVER;

                    hr=CoGetClassObject(CLSID_Koala, dwClsCtx, NULL
                        , IID_IClassFactory
                        , (LPLPVOID)&pIClassFactory);

                    if (SUCCEEDED(hr))
                        {
                        //Create the Koala by asking for IID_IPersist
                        pIClassFactory->CreateInstance(NULL
                            , IID_IPersist
                            , (LPLPVOID)&pAV->m_pIPersist);
```

(continued)

Listing 4-2. *continued*

```
                //Release the class factory when done.
                pIClassFactory->Release();
                }

            break;

        case IDM_OBJECTCREATECOCI:
            if (NULL!=pAV->m_pIPersist)
                {
                pAV->m_pIPersist->Release();
                pAV->m_pIPersist=NULL;
                CoFreeUnusedLibraries();
                }

            //Simpler creation: Use CoCreateInstance
            dwClsCtx=(pAV->m_fEXE) ? CLSCTX_LOCAL_SERVER
                : CLSCTX_INPROC_SERVER;

            CoCreateInstance(CLSID_Koala, NULL, dwClsCtx
                , IID_IPersist
                , (LPLPVOID)&pAV->m_pIPersist);

            break;

        case IDM_OBJECTRELEASE:
            if (NULL==pAV->m_pIPersist)
                break;

            pAV->m_pIPersist->Release();
            pAV->m_pIPersist=NULL;

            CoFreeUnusedLibraries();
            break;

        case IDM_OBJECTGETCLASSID:
            if (NULL==pAV->m_pIPersist)
                break;

            hr=pAV->m_pIPersist->GetClassID(&clsID);

            if (SUCCEEDED(hr))
                {
                LPSTR       psz;
                LPMALLOC    pIMalloc;
```

(continued)

Listing 4-2. *continued*

```
                    //String from CLSID uses task Malloc
                    StringFromCLSID(clsID, &psz);
                    MessageBox(hWnd, psz, "Object Class ID"
                        , MB_OK);

                    CoGetMalloc(MEMCTX_TASK, &pIMalloc);
                    pIMalloc->Free(psz);
                    pIMalloc->Release();
                    }
                else
                    {
                    MessageBox(hWnd
                        , "IPersist::GetClassID call failed"
                        , "Koala Demo", MB_OK);
                    }

                break;

            case IDM_OBJECTEXIT:
                PostMessage(hWnd, WM_CLOSE, 0, 0L);
                break;
            }
        break;

    default:
        return (DefWindowProc(hWnd, iMsg, wParam, lParam));
    }

return 0L;
}

CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev
    , UINT nCmdShow)
    {
    m_hInst        =hInst;
    m_hInstPrev    =hInstPrev;
    m_nCmdShow     =nCmdShow;

    m_hWnd         =NULL;
    m_fEXE         =FALSE;

    m_pIPersist    =NULL;
    m_fInitialized=FALSE;
    return;
    }
```

(continued)

Listing 4-2. *continued*

```
CAppVars::~CAppVars(void)
    {
    if (NULL!=m_pIPersist)
        m_pIPersist->Release();

    if (IsWindow(m_hWnd))
        DestroyWindow(m_hWnd);

    if (m_fInitialized)
        CoUninitialize();

    return;
    }

BOOL CAppVars::FInit(void)
    {
    WNDCLASS    wc;
    DWORD       dwVer;

    dwVer=CoBuildVersion();

    if (rmm!=HIWORD(dwVer))
        return FALSE;

    if (FAILED(CoInitialize(NULL)))
        return FALSE;

    m_fInitialized=TRUE;

    if (!m_hInstPrev)
        {
        wc.style          = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc    = ObjectUserWndProc;
        wc.cbClsExtra     = 0;
        wc.cbWndExtra     = CBWNDEXTRA;
        wc.hInstance      = m_hInst;
        wc.hIcon          = LoadIcon(m_hInst, "Icon");
        wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
        wc.lpszClassName  = "OBJUSER";

        if (!RegisterClass(&wc))
            return FALSE;
        }

    m_hWnd=CreateWindow("OBJUSER", "Koala Component Object Demo"
        , WS_OVERLAPPEDWINDOW, 35, 35, 350, 250, NULL, NULL
        , m_hInst, this);
```

(continued)

Listing 4-2. *continued*

```
    if (NULL==m_hWnd)
        return FALSE;

    ShowWindow(m_hWnd, m_nCmdShow);
    UpdateWindow(m_hWnd);

    CheckMenuItem(GetMenu(m_hWnd), IDM_OBJECTUSEDLL, MF_CHECKED);
    CheckMenuItem(GetMenu(m_hWnd), IDM_OBJECTUSEEXE, MF_UNCHECKED);

    return TRUE;
    }
```

OBJUSER's only interesting output is a message box that shows the CLSID retrieved from the object when you make a call to *IPersist::GetClassID*. Otherwise, you should step through this program in a debugger to really understand what is happening. In any case, the first two items on the Koala Object menu control whether you use the object implemented in an application or in a DLL. Either way, the rest of the functions remain the same. You can instantiate objects in one of two ways: You can use either *CoCreateInstance* or *CoGetClassObject* and *IClassFactory::CreateInstance*; you can also call the object's *Release* function and generate a call to *IPersist::GetClassID*, which displays the object's CLSID as a string in a message box. What a hot user interface!

Note that to run OBJUSER, you must have both compiled versions of the Koala object: DKOALA.DLL (CHAP04\DKOALA) and EKOALA.EXE (CHAP04\EKOALA). After you run MAKEALL.BAT for this chapter, both files will be in the BUILD directory on your disk. You must then let the Component Object library know where they are located by merging the CHAP04\CHAP04.REG file[2] with your current Registration Database using the Windows 3.1 RegEdit program. This will create entries for CLSID_Koala indicating where DKOALA.DLL and EKOALA.EXE are located. This is one of the powerful features of such registration: The object user is isolated from the need to locate the module that implements the object. The Registration Database essentially maps the CLSID to the path of the appropriate DLL or EXE.

For OBJUSER and any other component user, the following three steps instantiate and manage a component object. (Note that OBJUSER also performs the four steps outlined in the earlier section "The New Application for Windows Objects" because it's an EXE and defines a task.)

2. Note that this registration file does not contain full pathnames. Normally, all path entries in the Registration Database should contain full pathnames to modules. However, because there are so many modules to deal with in this book and because you might have installed them anywhere on your machine, the REG files given here do not include pathnames. That is why I recommended in Chapter 2 that you add the BUILD directory in the sample code to your PATH.

1. Use *#include <initguid.h>* in one source file of the compilation after including COMPOBJ.H to create a code segment containing CLSIDs and IIDs.
2. Create an object based on a CLSID using one of two routes:
   - If you need only one object, call *CoCreateInstance* with the CLSID and the IID of the interface you want on the object.
   - If you need more than one object, call *CoGetClassObject* to obtain a class factory (an *IClassFactory* pointer) for the CLSID, and call *IClassFactory::CreateInstance* as often as you want with the IID of the interface you want on the object. Call *IClassFactory::Release* when you have finished.
3. Use the object through the interface pointer, call *Release* through that pointer when finished, and call *CoFreeUnusedLibraries.*

The first step affects only your build environment and compilation, but it does not really matter in programming. The second step is the real meat of our discussion; it shows exactly how to instantiate a component object. The third step deals with how you manage and free the object through your interface pointer.

#include <initguid.h> and Precompiled Headers

Anything that ever references any GUID, be it a CLSID or an IID, must include the file INITGUID.H once, and only once, in the entire compilation of your module. This includes all component users and all objects (component objects or not) and means that you should use *#include <initguid.h>* in one, and only one, file of your application after including COMPOBJ.H. Including INITGUID.H ensures that all your GUIDs are defined and that they end up in a discardable code segment instead of in your data segment, which is preferred because defined GUIDs are always constant. INITGUID.H also allows you to use the DEFINE_GUID or DEFINE_OLEGUID macro for defining your own IDs as shown in the BOOKGUID.H file in the INC directory.

If you typically use a central include file for all files in your project, wrap an *#ifdef* statement around the *#include.* The samples in this book have such a statement in the shared BOOKGUID.H file in the INC directory:

```
#ifdef INITGUIDS
#include <initguid.h>
#endif
```

Only one file in each sample project uses *#define INITGUIDs.* Note that there is a similar symbol, INITGUID, used in COMPOBJ.H for similar purposes. However, you cannot use this symbol itself because COMPOBJ.H will not later pull in another necessary include file (COGUID.H)—that is, you will not be able to compile.

Including INITGUID.H only once is a trick when you are using precompiled headers, but you will appreciate it when we start including the lengthy OLE 2 header files. Create the precompiled header in a file that does not include INITGUID.H—the samples using precompilation all use the file PRECOMP.CPP, which contains only one *#include* statement. You can then use the precompiled header from this step to compile with all files *except* the one in which you *want* to include INITGUID.H. You should compile that single file without using the precompiled header to pull in the extra file.

Instantiate a Component Object

The Component Object library provides two fundamental object creation functions: *CoCreateInstance* and *CoGetClassObject* combined with *IClassFactory::CreateInstance.* Which functions you use depends on how many objects you need at a given time.

NOTE: Compound Document container applications do not directly use the *CoCreateInstance* or *CoGetClassObject* function to create compound document objects. Instead, they use functions such as *OleCreate* that internally use *CoCreateInstance,* as discussed in Chapter 9. If you plan to implement containers, you still should understand how compound document objects are created through these mechanisms.

To create a single object given a CLSID, use *CoCreateInstance,* which internally uses *CoGetClassObject,* as described a little later. The following code demonstrates this call and is adapted from OBJUSER, modifying the symbols and their locations for ease of explanation:

```
HRESULT     hr;
DWORD       dwClsCtx;
LPPERSIST   pIPersist;
LPUNKNOWN   pUnkOuter=NULL;

//fEXE controls where the object lives based on a menu selection.
dwClsCtx=(fEXE) ? CLSCTX_LOCAL_SERVER : CLSCTX_INPROC_SERVER;

CoCreateInstance(CLSID_Koala, pUnkOuter, dwClsCtx
    , IID_IPersist, (LPLPVOID)&pIPersist);
```

First note the naming of pointers to interfaces, as shown with LPPERSIST and LPUNKNOWN. OLE 2 follows a convention in which a far pointer to an interface of type *IInterface FAR ** is typed as LPINTERFACE—that is, the interface name sans *I* is appended in all caps to an *LP*. Thus LPUNKNOWN is an *IUnknown FAR ** and LPPERSIST is an *IPersist FAR **. In addition, note that *LPLPVOID* is defined in BOOKGUID.H as *LPVOID FAR **. *LPLPVOID* is simply a more convenient shorthand that I use in the code in this book.

CoCreateInstance takes five parameters, the names of which vary with the object class and interfaces you are using in your own implementation. Those shown here are similar to those in the OBJUSER program:

Parameter	Meaning
CLSID_Koala	REFIID: A reference (a real C++ reference) to the class identifier of the object you want to create. In this example, we are creating a Koala object implemented in the later section "Implementing a Component Object and Server." Note that in C, because there is no concept of a reference, you must precede this value with the *&* operator, thus: *&CLSID_Koala*.
pUnkOuter	LPUNKNOWN: A pointer to the controlling unknown if the object is being created as part of an aggregate. See the section "Object Reusability" later in this chapter for more information.
dwClsCtx	DWORD: Flags indicating the context in which the object is allowed to run, which can be any combination of CLSCTX-_LOCAL_SERVER (object in EXE), CLSCTX_INPROC-_SERVER (object in DLL), or CLSCTX_INPROC_HANDLER (object handler DLL). OBJUSER chooses to run either a DLL-based or an EXE-based object depending on a menu option stored in the *fExe* variable.
IID_IPersist	REFIID: A reference to the interface identifier you want to obtain for this object. If the object does not support this interface, *CoCreateInstance* will fail. Note that, as with the class identifier, C programs must again prepend the *&* operator to an *IID*.
&pIPersist	LPVOID FAR * (or LPLPVOID): A pointer to the location in which *CoCreateInstance* is to store the interface pointer on return. If *CoCreateInstance* fails, the contents of this variable will be set to NULL. Otherwise, *CoCreateInstance* will also call *AddRef* through the pointer before returning.

Note that when more than one CLSCTX_... flag is specified, the libraries will attempt to load them in the order CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, and then CLSCTX_LOCAL_SERVER;

that is, the libraries always look for a DLL first (for better performance), trying another EXE only as a last resort.

CoCreateInstance internally executes a three-step process to create the new object, which can be written in pseudocode as follows:

```
BEGIN
    Obtain a class factory (IClassFactory) for the desired class.
    Call IClassFactory::CreateInstance to create the object.
    Call IClassFactory::Release.
END
```

The first step, obtaining a *class factory* (also called a class object), is the exact purpose of the other relevant API function, *CoGetClassObject.* A class factory is an object that implements the *IClassFactory* interface, as defined in COMPOBJ.H:

```
DECLARE_INTERFACE_(IClassFactory, IUnknown)
    {
    [IUnknown methods included]

    //IClassFactory methods
    STDMETHOD(CreateInstance) (THIS_ IUnknown FAR* pUnkOuter, REFIID riid
        , LPVOID FAR * ppvObject) PURE;
    STDMETHOD(LockServer) (THIS_ BOOL fLock) PURE;
    };
```

For cases in which you want to create only one object of this class, *CoCreateInstance* suffices. However, if you want to create more than one object at a time, call *CoGetClassObject* to retrieve a class factory for the class, call *IClassFactory::CreateInstance* as many times as necessary, and call *IClassFactory::Release* when you have finished. The following code shows an implementation equivalent to the previous code but uses *CoGetClassObject* instead:

```
HRESULT             hr;
DWORD               dwClsCtx;
LPPERSIST           pIPersist;
LPUNKNOWN           pUnkOuter=NULL;
LPCLASSFACTORY      pIClassFactory;

dwClsCtx=(fEXE) ? CLSCTX_LOCAL_SERVER
    : CLSCTX_INPROC_SERVER;

hr=CoGetClassObject(CLSID_Koala, dwClsCtx, NULL
    , IID_IClassFactory, (LPLPVOID)&pIClassFactory);

if (SUCCEEDED(hr))
```

(continued)

```
    {
    //Create the Koala by asking for IID_IPersist
    pIClassFactory->CreateInstance(pUnkOuter
        , IID_IPersist, (LPLPVOID)&pIPersist);

    //We've finished with the class factory, so release it.
    pIClassFactory->Release();
    }
```

This code is almost the exact implementation of *CoCreateInstance* inside the component object library: The parameters you pass to *CoCreateInstance* are simply passed to *CoGetClassObject* and *IClassFactory::CreateInstance.* The extra parameters to *CoGetClassObject* are a NULL (a reserved LPVOID that should always be NULL), the interface ID you want on the class object (always *IID_IClassFactory* in OLE 2 but perhaps with more options in the future), and a location in which to store the pointer to the class object.

Remember that because *CoGetClassObject* is a function that creates a new interface pointer (reference counting rule #1 in the section "Reference Counting" in Chapter 3), you are responsible for calling *Release* through that pointer when you have finished, as shown in the preceding code.

NOTE: If you want to hold the class factory object for a longer time, you must call *IClassFactory::LockServer(TRUE).* A reference count on a class factory does not guarantee that the server will stay in memory and that you could use the class factory later. For these reasons, read the later section "Provide an Unloading Mechanism" under "Implementing a Component Object and Server." In short, if a class factory reference count could be used to keep a server in memory, the server could shut down only when its reference count reached zero. In the case of an EXE server, that reference can reach zero only if the server is being shut down. Catch-22. You must therefore use *LockServer(TRUE)* when you hold onto a class factory and *LockServer(FALSE)* after you release it. Besides, if the server is locked, retrieving another class factory is cheap.

Manage the Object and Call *CoFreeUnusedLibraries*

What you do with an object after you have obtained an interface pointer is entirely dependent on the object itself and is really what most of the chapters in the book are about. You must, in any case, be absolutely sure to call *Release* through that interface pointer when you have finished with the object. Otherwise, you doom the object to live in memory for all eternity—or until the universe collapses (power off or the jolly three-finger reset).

Releasing the object is not the only consideration, however. When you initially instantiate an object implemented in a DLL, COMPOBJ.DLL loads that DLL into memory using the function *CoLoadLibrary*. When the DLL is no longer needed, COMPOBJ.DLL calls *CoFreeLibrary*. Both functions map to *LoadLibrary* and *FreeLibrary* under Windows but are named differently for portability to other platforms, such as the Apple Macintosh.

However, the Component Object library does not know when an object in a DLL is destroyed because once it has facilitated loading and instantiating that object, communication between the component object and the component user is direct, completely bypassing everything in the library. Therefore, a DLL might remain loaded in memory even when it has no objects to service. Over time, many DLLs might be loaded and chew up valuable memory. The Component Object library needs a cue to free those DLLs that are no longer needed. This is very much like discardable global memory, in which memory allocated and freed will stay in memory until discarded, even if no one is using that memory.

For this reason, all object users should periodically call *CoFreeUnusedLibraries*, primarily immediately after you use *Release* on an object for good. In this function, COMPOBJ.DLL can ask each DLL loaded in your task whether it can be unloaded. If the DLL answers "yes," *CoFreeLibrary* is called to free the memory the DLL occupies. *CoFreeUnusedLibraries* does not, however, affect other EXEs because EXEs unload themselves when they are no longer servicing any objects. Using *CoFreeUnusedLibraries* is something like calling *GlobalCompact(-1)*, which will purge memory of all unreferenced discardable memory segments.

NOTE: The OLE 2 implementation of *CoFreeUnusedLibraries* does nothing. However, your user code should still call the function after destroying an object so that when the function is implemented, your code will work correctly.

Implementing a Component Object and a Server

Let's now implement a simple Koala Component Object with the *IPersist* interface, where just the object itself is shown in the source code in Listing 4-3. Koala implements the *IPersist* interface because that interface has standard OLE 2–provided marshaling support, meaning that we can freely place this object in a DLL or an EXE, as we'll actually demonstrate. In real-world use, *IPersist* is never implemented alone because it always serves as a base class for a few other interfaces.

KOALA.H

```
/*
 * Koala Object DLL/EXE Chapter 4
 *
 * Classes that implement the Koala object independent of whether
 * we live in a DLL or an EXE
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _KOALA_H_
#define _KOALA_H_

#include <windows.h>
#include <ole2.h>          //ole2.h has IPersist, compobj.h doesn't

#include <bookguid.h>

//Type for an object-destroyed callback
typedef void (FAR PASCAL *LPFNDESTROYED)(void);

//Forward class references
class __far CImpIPersist;
typedef class CImpIPersist FAR *LPIMPIPERSIST;

/*
 * The Koala object is implemented in its own class with its own
 * IUnknown to support aggregation. It contains one CImpIPersist
 * object that we use to implement the externally exposed interfaces.
 */

class __far CKoala : public IUnknown
    {
    //Make any contained interfaces friends
    friend class CImpIPersist;

    protected:
        ULONG           m_cRef;          //Object reference count
        LPUNKNOWN       m_pUnkOuter;     //Controlling unknown
        LPFNDESTROYED   m_pfnDestroy;    //Function closure call
        LPIMPIPERSIST   m_pIPersist;     //Contained interface
```

Listing 4-3.

(continued)

Implementation of the Koala object structured to live in either a DLL or an EXE.

Listing 4-3. *continued*

```
    public:
        CKoala(LPUNKNOWN, LPFNDESTROYED);
        ~CKoala(void);

        BOOL FInit(void);

        //Non-delegating object IUnknown
        STDMETHODIMP         QueryInterface(REFIID, LPLPVOID);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);
    };

typedef CKoala FAR *LPCKoala;

/*
 * Interface implementations for the CKoala object.
 */

class __far CImpIPersist : public IPersist
    {
    private:
        ULONG           m_cRef;
        LPCKoala        m_pObj;         //Back pointer to the object
        LPUNKNOWN       m_punkOuter;    //Controlling unknown

    public:
        CImpIPersist(LPCKOALA, LPUNKNOWN);
        ~CImpIPersist(void);

        //IUnknown members that delegate to m_pUnkOuter.
        STDMETHODIMP         QueryInterface(REFIID, LPLPVOID);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);

        //IPersist members
        STDMETHODIMP         GetClassID(LPCLSID);
    };

#endif //_KOALA_H_
```

(continued)

Listing 4-3. *continued*

KOALA.CPP

```
/*
 * Koala Object DLL/EXE Chapter 4
 *
 * Implementation of the CKoala and CImpIPersist objects that works
 * in either an EXE or a DLL
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 *
 */

#include "koala.h"

CKoala::CKoala(LPUNKNOWN pUnkOuter, LPFNDESTROYED pfnDestroy)
    {
    m_cRef=0;
    m_pUnkOuter=pUnkOuter;
    m_pfnDestroy=pfnDestroy;

    //NULL any contained interfaces initially.
    m_pIPersist=NULL;

    return;
    }

CKoala::~CKoala(void)
    {
    //Free contained interfaces.
    if (NULL!=m_pIPersist)
        delete m_pIPersist;     //Interface does not free itself.

    return;
    }

BOOL CKoala::FInit(void)
    {
    LPUNKNOWN       pIUnknown=(LPUNKNOWN)this;

    if (NULL!=m_pUnkOuter)
        pIUnknown=m_pUnkOuter;

    //Allocate contained interfaces.
    m_pIPersist=new CImpIPersist(this, pIUnknown);
```

(continued)

Listing 4-3. *continued*

```
    return (NULL!=m_pIPersist);
    }

STDMETHODIMP CKoala::QueryInterface(REFIID riid, LPLPVOIDppv)
    {
    *ppv=NULL;

    /*
     * The only calls for IUnknown are either in a nonaggregated
     * case or when created in an aggregation, so in either case,
     * always return our IUnknown for IID_IUnknown.
     */
    if (IsEqualIID(riid, IID_IUnknown))
        *ppv=(LPVOID)this;

    /*
     * For IPersist, we return our contained interface. For EXEs, we
     * have to return our interface for IPersistStorage as well
     * since OLE 2 doesn't support IPersist implementations by
     * themselves (assumed only to be a base class). If a user
     * asked for an IPersistStorage and used it, they would crash--
     * but this is a demo, not a real object.
     */
    if (IsEqualIID(riid, IID_IPersist)
        || IsEqualIID(riid, IID_IPersistStorage))
        *ppv=(LPVOID)m_pIPersist;

    //AddRef any interface we'll return.
    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    return ResultFromScode(E_NOINTERFACE);
    }

STDMETHODIMP_(ULONG) CKoala::AddRef(void)
    {
    return ++m_cRef;
    }

STDMETHODIMP_(ULONG) CKoala::Release(void)
    {
```

(continued)

Listing 4-3. *continued*

```
    ULONG       cRefT;

    cRefT=--m_cRef;

    if (0==m_cRef)
        {
        /*
         * Tell the housing that an object is going away so that it
         * can shut down if appropriate.
         */
        if (NULL!=m_pfnDestroy)
            (*m_pfnDestroy)();

        delete this;
        }

    return cRefT;
    }

CImpIPersist::CImpIPersist(LPCKoala pObj, LPUNKNOWN pUnkOuter)
    {
    m_cRef=0;
    m_pObj=pObj;
    m_pUnkOuter=pUnkOuter;
    return;
    }

CImpIPersist::~CImpIPersist(void)
    {
    return;
    }

STDMETHODIMP CImpIPersist::QueryInterface(REFIID riid
    , LPVOID FAR *ppv)
    {
    return m_pUnkOuter->QueryInterface(riid, ppv);
    }

STDMETHODIMP_(ULONG) CImpIPersist::AddRef(void)
    {
    ++m_cRef;
    return m_pUnkOuter->AddRef();
    }
```

(continued)

Listing 4-3. *continued*

```
STDMETHODIMP_(ULONG) CImpIPersist::Release(void)
    {
    --m_cRef;
    return m_pUnkOuter->Release();
    }

STDMETHODIMP CImpIPersist::GetClassID(LPCLSID pClsID)
    {
    *pClsID=CLSID_Koala;
    return NOERROR;
    }
```

The Koala object is implemented to support aggregation and to be identically usable in either a DLL or an EXE server, but not without some impact. In addition, remember that we can call *CoGetMalloc* at any time to obtain access to shared or task memory, although the implementation of Koala shown here does not have occasion to use this feature.

To support aggregation, as we'll see in the later section "Object Reusability," the object must be aware of a *controlling unknown*, if there is one, that is cognizant of all interfaces supported by the aggregate object. The object itself, implemented by using the *CKoala* C++ class,[3] implements *IUnknown* but contains the *interface implementation* of *IPersist* in another C++ object named *CImpIPersist*. In aggregation, the interface implementations must always delegate all their *IUnknown* calls to the object that controls that interface's lifetime. When the object is not aggregated and when the *pUnkOuter* passed to *CKoala::CKoala* is NULL, *CKoala* passes its own *IUnknown* implementation to *CImpIPersist* instead. When the Koala object is aggregated, *CKoala* will receive a non-NULL *pUnkOuter*, which it passes to *CImpIPersist*. In either case, the *IPersist* interface implementation will always delegate *IUnknown* calls to a full object, performing only trivial reference counting on the interface for debugging purposes. If this seems confusing, be patient; we'll see this in more detail later.

The only interesting function of *CImpIPersist* is *GetClassID*, which simply returns the CLSID defined by the Koala object. However, the implementation of *CKoala*, the entire object, has a few more interesting features. First, note that although we hold onto a copy of the *pUnkOuter* pointer, we do *not* call *AddRef* through it. We do this to avoid a problem with circular reference counts: If we did call *AddRef* on *pUnkOuter*, the outer object could not free

3. Remember that a C++ class is a *convenient* way to create interface function tables. *CKoala* is never exposed to anything outside its DLL or EXE, period. It exposes only its *IUnknown* and *IPersist* interface function tables.

itself unless our object is freed first. But the outer object will not free our object unless it's freeing itself. The solution to this conundrum is to realize that our object's lifetime is entirely contained within the lifetime of the outer object, so we don't have to make the extra *AddRef* call, and at the same time avoid the circular reference count.

Second, we supply a two-phase instantiation process for use by the class factory we provide later. The *CKoala* constructor initializes only variables, whereas *FInit* performs any operations that are prone to failure, so the caller can determine whether a failure did occur. Because instantiating the *CImpIPersist* interface implementation might fail, we defer that action until *FInit* is called.

Next, the *QueryInterface* implementation in *CKoala*, which knows all the interfaces implemented in this object, makes a special case for the *interface* called *IPersistStorage*. When a user such as OBJUSER asks the DLL implementation for an interface identified with *IID_IPersist*, that IID comes directly into *QueryInterface*. However, when OBJUSER asks for *IID_IPersist* and the object lives in an EXE, that request goes through the marshaling layer in COMPOBJ.DLL. The OLE 2 implementation of this marshaling does not single out *IPersist* and will always ask the object for *IPersistStorage* even if the user asked only for *IPersist*. So we also check for *IPersistStorage* here. Of course, you must avoid this in real applications because the user might actually have asked for *IPersistStorage* but received only an *IPersist*. But as I pointed out earlier, *IPersist* is never useful when implemented alone—it's used here only for demonstration.

The final feature of the Koala object allows it to notify its server—either a DLL or an EXE—when the object is destroyed in *Release* by calling an "object destroyed" function in the server. This is a special technique I created to isolate the object from any specifics about its DLL or EXE server—it's not part of the OLE 2 specifications. When the object is created, the class factory passes the address of the "object destroyed" function to the object. In the case of both DKOALA and EKOALA, the function is named (what else?) *ObjectDestroyed*, and a pointer to that function is of type LPFNDESTROYED. (See KOALA.H.) When the object frees itself in its *Release* function, it calls *ObjectDestroyed*, in which the server decrements the count of objects it's currently servicing. If the object lives in a DLL, that DLL might be able to then mark itself as unloadable: if the object lives in an EXE and it was the last object, the EXE might shut down on this notification. This technique effectively lets the server worry about unloading and shutting down, keeping the object isolated. Of course, this is my technique—feel free to create your own.

With the object isolated from any concern about where it lives, we can now concentrate on seeing how you expose that object from a DLL or an EXE, which share these four steps to manage an object although their exact implementations of each step differ:

1. Register the CLSIDs and server pathnames for every class implemented in the server in the Registration Database.
2. Implement the class factory for each object class supported by the server. A single DLL or EXE server can handle any number of classes.
3. Expose the class factory to the Component Object library.
4. Provide a shutdown or an unloading mechanism when there are no more objects or lock counts on the server.

The DLL housing of Koala, DKOALA.DLL, is shown in Listing 4-4, and the EXE housing, EKOALA.EXE, is shown in Listing 4-5. Note that the Koala object implementation itself, shown in Listing 4-3, is identical in both the CHAP04\DKOALA and CHAP04\EKOALA directories in the sample code. When you run the OBJUSER program using each of these servers, you will notice a difference in the response time of calling the object's *GetClassID*. When you are using a DLL server, the response is quick because the call goes directly to the object implementation. When you are using an EXE server, the response is slower because the *GetClassID* call must be worked through the marshaling process.

DKOALA.H

```
/*
 * Koala Object DLL Chapter 4
 *
 * Definitions, classes, and prototypes for a DLL that
 * provides Koala objects to any other object user.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _DKOALA_H_
#define _DKOALA_H_

//Get the object definitions
#include "koala.h"

void FAR PASCAL ObjectDestroyed(void);

//DKOALA.CPP
VOID FAR PASCAL WEP(int);

//This class factory object creates Koala objects.
```

(continued)

Listing 4-4.
The DKOALA.DLL implementation to house the Koala object.

Listing 4-4. *continued*

```
class __far CKoalaClassFactory : public IClassFactory
    {
    protected:
        ULONG           m_cRef;

    public:
        CKoalaClassFactory(void);
        ~CKoalaClassFactory(void);

        //IUnknown members
        STDMETHODIMP         QueryInterface(REFIID, LPLPVOID);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);

        //IClassFactory members
        STDMETHODIMP         CreateInstance(LPUNKNOWN, REFIID
                                 , LPLPVOID);
        STDMETHODIMP         LockServer(BOOL);
    };

typedef CKoalaClassFactory FAR *LPCKoalaClassFactory;

#endif //_DKOALA_H_
```

DKOALA.CPP

```
/*
 * Koala Object DLL Chapter 4
 *
 * Example object implemented in a DLL. This object supports
 * IUnknown and IPersist interfaces; it doesn't know anything more
 * than how to return its class ID, but it demonstrates how an
 * object is presented inside a DLL.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

//Do this once in the entire build
#define INITGUIDS

#include "dkoala.h"

//Count number of objects and number of locks.
ULONG       g_cObj=0;
ULONG       g_cLock=0;
```

(continued)

Listing 4-4. *continued*

```
[LibMain and WEP omitted from listing]

HRESULT EXPORT FAR PASCAL DllGetClassObject(REFCLSID rclsid
    , REFIID riid, LPVOID FAR *ppv)
    {

    if (!IsEqualCLSID(rclsid, CLSID_Koala))
        return ResultFromScode(E_FAIL);

    //Check that we can provide the interface
    if (!IsEqualIID(riid, IID_IUnknown)
        && !IsEqualIID(riid, IID_IClassFactory))
        return ResultFromScode(E_NOINTERFACE);

    //Return our IClassFactory for Koala objects
    *ppv=(LPVOID)new CKoalaClassFactory();

    if (NULL==*ppv)
        return ResultFromScode(E_OUTOFMEMORY);

    //AddRef the object through any interface we return
    ((LPUNKNOWN)*ppv)->AddRef();

    return NOERROR;
    }

STDAPI DllCanUnloadNow(void)
    {
    SCODE   sc;

    //Our answer is whether there are any object or locks
    sc=(0L==g_cObj && 0==g_cLock) ? S_OK : S_FALSE;
    return ResultFromScode(sc);
    }

/*
 * ObjectDestroyed
 *
 * Purpose:
 *  Function for the Koala object to call when it is destroyed.
 *  Because we're in a DLL, we only track the number of objects here,
 *  letting DllCanUnloadNow take care of the rest.
```

(continued)

Listing 4-4. *continued*

```
 */

void FAR PASCAL ObjectDestroyed(void)
    {
    g_cObj--;
    return;
    }

CKoalaClassFactory::CKoalaClassFactory(void)
    {
    m_cRef=0L;
    return;
    }

CKoalaClassFactory::~CKoalaClassFactory(void)
    {
    return;
    }

STDMETHODIMP CKoalaClassFactory::QueryInterface(REFIID riid
    , LPLPVOID ppv)
    {
    *ppv=NULL;

    //Any interface on this object is the object pointer.
    if (IsEqualIID(riid, IID_IUnknown)
        || IsEqualIID(riid, IID_IClassFactory))
        *ppv=(LPVOID)this;

    /*
     * If we actually assign an interface to ppv we need to AddRef
     * it because we're returning a new pointer.
     */
    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    return ResultFromScode(E_NOINTERFACE);
    }

STDMETHODIMP_(ULONG) CKoalaClassFactory::AddRef(void)
    {
    return ++m_cRef;
```

(continued)

Listing 4-4. *continued*

```
    }

STDMETHODIMP_(ULONG) CKoalaClassFactory::Release(void)
    {
    ULONG           cRefT;

    cRefT=--m_cRef;

    if (0L==m_cRef)
        delete this;

    return cRefT;
    }

STDMETHODIMP CKoalaClassFactory::CreateInstance(LPUNKNOWN pUnkOuter
    , REFIID riid, LPVOID FAR *ppvObj)
    {
    LPCKoala            pObj;
    HRESULT             hr;

    *ppvObj=NULL;
    hr=ResultFromScode(E_OUTOFMEMORY);

    //Verify that a controlling unknown asks for IUnknown
    if (NULL!=pUnkOuter && !IsEqualIID(riid, IID_IUnknown))
        return ResultFromScode(E_NOINTERFACE);

    //Create the object, passing function to notify on destruction
    pObj=new CKoala(pUnkOuter, ObjectDestroyed);

    if (NULL==pObj)
        return hr;

    if (pObj->FInit())
        hr=pObj->QueryInterface(riid, ppvObj);

    //Kill the object if initial creation or FInit failed.
    if (FAILED(hr))
        delete pObj;
    else
        g_cObj++;

    return hr;
    }

STDMETHODIMP CKoalaClassFactory::LockServer(BOOL fLock)
```

(continued)

Listing 4-4. *continued*

```
    {
    if (fLock)
        g_cLock++;
    else
        g_cLock--;

    return NOERROR;
    }
```

EKOALA.H

```
/*
 * Koala Object Chapter 4
 *
 * Definitions, classes, and prototypes for an application that
 * provides Koala objects to any other object user.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _EKOALA_H_
#define _EKOALA_H_

//Get the object definitions
#include "koala.h"

//EKOALA.CPP
LRESULT FAR PASCAL EXPORT KoalaWndProc(HWND, UINT, WPARAM, LPARAM);

class __far CAppVars
    {
    friend LRESULT FAR PASCAL EXPORT KoalaWndProc(HWND, UINT
        , WPARAM, LPARAM);

    protected:
        HINSTANCE       m_hInst;            //WinMain parameters
        HINSTANCE       m_hInstPrev;
        LPSTR           m_pszCmdLine;
        UINT            m_nCmdShow;

        HWND            m_hWnd;             //Main window handle
```

(continued)

Listing 4-5.
The EKOALA.EXE implementation to house the Koala object.

Listing 4-5. *continued*

```
        BOOL            m_fInitialized;     //Did CoInitialize work?
        LPCLASSFACTORY  m_pIClassFactory;   //Our class factory
        DWORD           m_dwRegCO;          //Registration key

    public:
        CAppVars(HINSTANCE, HINSTANCE, LPSTR, UINT);
        ~CAppVars(void);
        BOOL FInit(void);
    };

typedef CAppVars FAR LPAPPVARS;

#define CBWNDEXTRA              sizeof(LONG)
#define KOALAWL_STRUCTURE       0

void FAR PASCAL ObjectDestroyed(void);

//This class factory object creates Koala objects.

class __far CKoalaClassFactory : public IClassFactory
    {
    protected:
        ULONG           m_cRef;

    public:
        CKoalaClassFactory(void);
        ~CKoalaClassFactory(void);

        //IUnknown members
        STDMETHODIMP         QueryInterface(REFIID, LPLPVOID);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);

        //IClassFactory members
        STDMETHODIMP         CreateInstance(LPUNKNOWN, REFIID
                                 , LPLPVOID);
        STDMETHODIMP         LockServer(BOOL);
    };

typedef CKoalaClassFactory FAR *LPCKoalaClassFactory;

#endif //_EKOALA_H_
```

(continued)

Listing 4-5. *continued*

EKOALA.CPP

```
/*
 * Koala Object EXE Chapter 4
 *
 * Object implemented in an application. This object supports
 * IUnknown and IPersist interfaces; it doesn't know anything more
 * than how to return its class ID, but it demonstrates how an
 * object is presented inside an EXE.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

//Do this once in the entire build
#define INITGUIDS

#include <ole2ver.h>
#include "ekoala.h"

//Count number of objects and number of locks.
ULONG       g_cObj=0;
ULONG       g_cLock=0;

//Make window handle global so that other code can cause a shutdown
HWND        g_hWnd=NULL;

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    MSG         msg;
    LPAPPVARS   pAV;
    int         cMsg=96;

   #ifndef WIN32
    //Enlarge the queue as large as we can starting from 96
    while (!SetMessageQueue(cMsg) && (cMsg-=8));
   #endif

    pAV=new CAppVars(hInst, hInstPrev, pszCmdLine, nCmdShow);

    if (NULL==pAV)
        return -1;

    if (pAV->FInit())
```

(continued)

Listing 4-5. *continued*

```
        {
        while (GetMessage(&msg, NULL, 0, 0))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    delete pAV;
    return msg.wParam;
    }

LRESULT FAR PASCAL EXPORT KoalaWndProc(HWND hWnd, UINT iMsg
    , WPARAM wParam, LPARAM lParam)
    {
    LPAPPVARS   pAV;

    pAV=(LPAPPVARS)GetWindowLong(hWnd, KOALAWL_STRUCTURE);

    switch (iMsg)
        {
        case WM_NCCREATE:
            pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)
                ->lpCreateParams);

            SetWindowLong(hWnd, KOALAWL_STRUCTURE, (LONG)pAV);
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
        }

    return 0L;
    }

/*
 * ObjectDestroyed
 *
 * Purpose:
 *  Function for the Koala object to call when it gets destroyed.
 *  We destroy the main window if the proper conditions are met for
 *  shutdown.
```

(continued)

Listing 4-5. *continued*

```
 */

void FAR PASCAL ObjectDestroyed(void)
    {
    g_cObj--;

    //No more objects and no locks, shut the app down.
    if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
        PostMessage(g_hWnd, WM_CLOSE, 0, 0L);

    return;
    }

CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, UINT nCmdShow)
    {
    //Initialize WinMain parameter holders.
    m_hInst     =hInst;
    m_hInstPrev =hInstPrev;
    m_pszCmdLine=pszCmdLine;
    m_nCmdShow  =nCmdShow;

    m_hWnd=NULL;
    m_dwRegCO=0;
    m_pIClassFactory=NULL;
    m_fInitialized=FALSE;
    return;
    }

CAppVars::~CAppVars(void)
    {
    //Opposite of CoRegisterClassObject; class factory ref is now 1
    if (0L!=m_dwRegCO)
        CoRevokeClassObject(m_dwRegCO);

    //This should be the last Release, which frees the class factory.
    if (NULL!=m_pIClassFactory)
        m_pIClassFactory->Release();

    if (m_fInitialized)
        CoUninitialize();

    return;
    }
```

(continued)

Listing 4-5. *continued*

```
BOOL CAppVars::FInit(void)
    {
    WNDCLASS    wc;
    HRESULT     hr;
    DWORD       dwVer;

    //Fail if we've run outside CoGetClassObject
    if (lstrcmpi(m_pszCmdLine, "-Embedding"))
        return FALSE;

    dwVer=CoBuildVersion();

    if (rmm!=HIWORD(dwVer))
        return FALSE;

    if (FAILED(CoInitialize(NULL)))
        return FALSE;

    m_fInitialized=TRUE;

    if (!m_hInstPrev)
        {
        wc.style          = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc    = KoalaWndProc;
        wc.cbClsExtra     = 0;
        wc.cbWndExtra     = CBWNDEXTRA;
        wc.hInstance      = m_hInst;
        wc.hIcon          = NULL;
        wc.hCursor        = NULL;
        wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName   = NULL;
        wc.lpszClassName  = "Koala";

        if (!RegisterClass(&wc))
            return FALSE;
        }

    m_hWnd=CreateWindow("Koala", "Koala", WS_OVERLAPPEDWINDOW
        , 35, 35, 350, 250, NULL, NULL, m_hInst, this);

    if (NULL==m_hWnd)
        return FALSE;

    g_hWnd=m_hWnd;
```

(continued)

Listing 4-5. *continued*

```
    /*
     * Create our class factory and register it for this application
     * using CoRegisterClassObject. We are able to service more than
     * one object at a time so we use REGCLS_MULTIPLEUSE.
     */
    m_pIClassFactory=new CKoalaClassFactory();

    if (NULL==m_pIClassFactory)
        return FALSE;

    //Because we hold on to this, we should AddRef it.
    m_pIClassFactory->AddRef();

    hr=CoRegisterClassObject(CLSID_Koala
        , (LPUNKNOWN)m_pIClassFactory, CLSCTX_LOCAL_SERVER
        , REGCLS_MULTIPLEUSE, &m_dwRegCO);

    if (FAILED(hr))
        return FALSE;

    return TRUE;
    }

CKoalaClassFactory::CKoalaClassFactory(void)
    {
    m_cRef=0L;
    return;
    }

CKoalaClassFactory::~CKoalaClassFactory(void)
    {
    return;
    }

STDMETHODIMP CKoalaClassFactory::QueryInterface(REFIID riid
    , LPVOID FAR *ppv)
    {
    *ppv=NULL;

    //Any interface on this object is the object pointer.
    if (IsEqualIID(riid, IID_IUnknown)
        || IsEqualIID(riid, IID_IClassFactory))
        *ppv=(LPVOID)this;

    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
```

(continued)

Listing 4-5. *continued*

```
            return NOERROR;
            }

        return ResultFromScode(E_NOINTERFACE);
        }

STDMETHODIMP_(ULONG) CKoalaClassFactory::AddRef(void)
        {
        return ++m_cRef;
        }

STDMETHODIMP_(ULONG) CKoalaClassFactory::Release(void)
        {
        ULONG           cRefT;

        cRefT=--m_cRef;

        if (0L==m_cRef)
            delete this;

        return cRefT;
        }

STDMETHODIMP CKoalaClassFactory::CreateInstance(LPUNKNOWN pUnkOuter
        , REFIID riid, LPVOID FAR *ppvObj)
        {
        LPCKoala            pObj;
        HRESULT             hr;

        *ppvObj=NULL;
        hr=ResultFromScode(E_OUTOFMEMORY);

        //Verify that a controlling unknown asks for IUnknown
        if (NULL!=pUnkOuter && !IsEqualIID(riid, IID_IUnknown))
            return ResultFromScode(E_NOINTERFACE);

        //Create the object, telling it to notify us when it's gone.
        pObj=new CKoala(pUnkOuter, ObjectDestroyed);

        if (NULL==pObj)
            {
            //Starts shutdown if no other objects
            g_cObj++;
            ObjectDestroyed();
            return hr;
            }
```

(continued)

Listing 4-5. *continued*

```
    if (pObj->FInit())
        hr=pObj->QueryInterface(riid, ppvObj);

    g_cObj++;

    /*
     * Kill the object if initial creation or FInit failed. If
     * the object failed, we handle the g_cObj increment above
     * in ObjectDestroyed.
     */
    if (FAILED(hr))
        {
        delete pObj;
        ObjectDestroyed();  //Handle shutdown cases.
        }

    return hr;
    }

STDMETHODIMP CKoalaClassFactory::LockServer(BOOL fLock)
    {
    if (fLock)
        g_cLock++;
    else
        {
        g_cLock--;

        //No more objects and no locks, shut the app down.
        if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
            PostMessage(g_hWnd, WM_CLOSE, 0, 0L);
        }

    return NOERROR;
    }
```

Register CLSIDs

Every component object class (but not all types of Windows Objects) must have a unique CLSID associated with it in the Registration Database. The registration entries for a simple object, such as we're implementing here, are few; as you create objects with more features that support linking and embedding, there will be much more information to add, as described in Chapter 10. For purposes of the sample code, the necessary entries are contained in the file CHAP04\CHAP04.REG, which you can add to your Registration Database by using the REGEDIT program in Windows. Creating a REG file is the

preferred method of registering objects and applications because it can be done at install time instead of programmatically at run-time, which is tedious, to say the least. The online help in the OLE 2 SDK contains more information about storing information in the Registration Database.

The required entries fall under the CLSID key, where OLE 2 stores information about all classes under your spelled-out CLSID, as you can see in the REGEDIT program and as shown in Figure 4-5 on the next page. OLE 2 also stores information about its standard interfaces and the code that handles parameter marshaling under the Interface key. The following steps describe the necessary registration for DLL-based and EXE-based objects:

1. From HKEY_CLASSES_ROOT (the root key of the entire Registration Database), create the entry *CLSID\{class ID}=<name>*, where {*class ID*} is the value of your CLSID spelled out and *<name>* is a human-readable string for your object. The Koala object has the class ID string {00021102-0000-0000-C000-000000000046} which is not something many, except for a few odd individuals, consider readable. The *<name>* of the Koala object is "Koala Object Chapter 4."

2. Create an entry under the CLSID entry in step 1 to point to the object code:

   - For DLL objects, register *InprocServer=<path to DLL>*.
   - For EXE objects, register *LocalServer=<path to EXE>*.
   - For DLL object handlers, register *InprocHandler=<path to DLL>*.

   Note that these entries should always contains full pathnames so that you do not depend on your DLLs or EXEs being on the MS-DOS path. Your application's install program should update the paths when it knows where the installation occurred.[4]

3. (Optional) If you want to allow a user to look up your CLSID based on a text string, make an entry under HKEY_CLASSES_ROOT of *<ProgID>=<name>*, where *<ProgID>* is a short name without spaces or punctuation, and *<name>* is the human-readable name, identical to that in step 1, of your object. Under this key, create another entry, CLSID=*{class ID}*, in which *{class ID}* is also the same as in step 1. In

4. The sample code with this book does, however, break this rule because the installation program on the companion disks is not capable of modifying all the .REG files in each CHAPxx directory to contain a full pathname. Instead, each DLL and EXE is registered without a full pathname and therefore depends on them being in the path.

this example, *<ProgID>* is *Koala* and *<name>* is "Koala Object Chapter 4." Note that you can also create a symmetric key under the object's CLSID in the form of *ProgID = <PRogID>*.

Entries of the type created in step 3 will be required for Compound Document objects that should appear in the Insert Object dialog box inside a container application. But that is a subject for a later chapter. Without those entries from steps 1 and 2, however, the *CoGetClassObject* API function (which *CoCreateInstance* uses, remember) will not be able to locate your object implementation. Note also that the same DLL or EXE can serve multiple CLSIDs, and in such cases you must make a similar entry under each CLSID you support with the *InprocServer* and *LocalServer* keys, although they can all contain the same path to the same server.

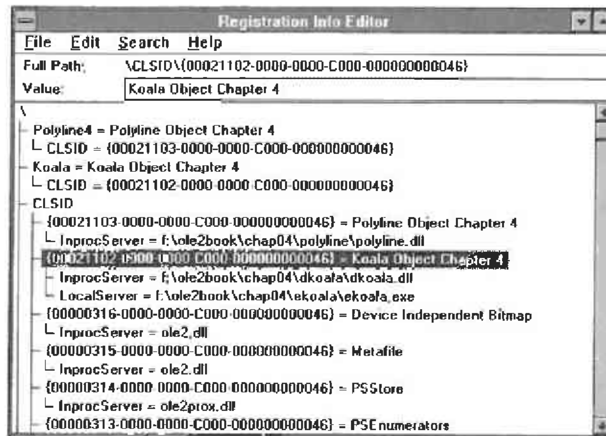


Figure 4-5.
The populated CLSID section of the Registration Database, showing the entries for Koala.

Implement the Class Factory

Telling the Component Object library where your object code lives is one thing; you still need to provide for creating objects once that code has been loaded, so the next step is to create a class factory that implements the *IClassFactory* interface. After we implement this class factory, we can provide the code to expose it outside the server. This is somewhat like implementing a window procedure in order to call *RegisterClass* because you have to store a pointer to your window procedure in the WNDCLASS before you call *RegisterCall.* Both sample implementations, DLL and EXE, use a C++ class of *CKoalaClassFactory* for this purpose, and the two are almost identical. The

only differences between the DLL and EXE implementations of *CKoalaClassFactory* have to do with the unloading mechanisms, which we'll discuss later. For now, let's concentrate on those identical parts that instantiate a *CKoala* object.

All implementations of *IClassFactory::CreateInstance* are identical, and each implementation contains three major points of interest. First, the first parameter to *CreateInstance* is *pUnkOuter*, which is the controlling unknown for the object we've been asked to create, if our new object is becoming part of an aggregate. When we instantiate the object using *new CKoala*, we pass this *pUnkOuter* down to the object so that it can delegate properly. (Again, see "Object Reusability" for more details.) When an object is aggregated, the outer object *must* ask for an *IUnknown* interface on the new object. To enforce this rule, we check that *IID_IUnknown* is asked for when *pUnkOuter* is not NULL.

Next, in addition to passing *pUnkOuter* to *new CKoala*, we also pass a pointer to an independent function named *ObjectDestroyed*. When the final call to *Release* for the Koala object is about to free the object, it will call this function. This allows the object to isolate itself from the nature of its server housing (DLL or EXE) and allow that housing to act appropriately on the event. You can see again in the *CKoala::Release* function in Listing 4-3 how and when this function is called. We'll examine what *ObjectDestroyed* does in both servers in the later section "Provide an Unloading Mechanism."

Finally, if the object is successfully instantiated, we still need to initialize it through an internal *FInit* implemented in the *CKoala* object. *FInit* is not a standard feature of OLE 2 objects and is used here to support a convenient two-phase creation model common in C++ coding. *FInit* performs all operations that might fail and is thus able to communicate success or failure back to the class factory during instantiation. If this second initialization step succeeds, *CreateInstance* asks the object's *QueryInterface* to return the appropriate interface pointer, which has the convenient effect of calling that pointer's *AddRef* as required. Remember that *CreateInstance*, as a function returning a new pointer, must return the pointer with a reference count on the caller's behalf. Furthermore, *CreateInstance* increments a global object count, which the server can use to determine unloading conditions. If the initialization fails, *CreateInstance* deletes the object and returns an out-of-memory error to the caller.

IClassFactory also has a member named *LockServer*, which either increments or decrements a lock count on the DLL or EXE in which the class factory lives. *LockServer* provides a method through which a user can keep a DLL or an EXE in memory even if that server is not servicing any objects and has no outstanding reference counts on its class factory. This allows a user to

optimize loading and reloading of servers, keeping the code in memory even when it's not immediately necessary. Such optimizations can greatly increase performance when a user deals with a very large server EXE or DLL.

The implementation of *LockServer* in the DLL and EXE versions differs slightly, again to handle the differences in unloading mechanisms (although we could isolate these differences as well). Their commonality is to either increment or decrement a global lock counter, which is used in different ways by the server's unloading mechanism.

Expose the Class Factory

The major difference between DLL and EXE servers is in how they expose their class factories, primarily because an EXE defines a task, whereas a DLL doesn't. Your class factory is an object, and the Component Object library needs to obtain its *IClassFactory* pointer. It does so either by calling a function that you export from an object DLL (that is, an API function that you implement) or by requiring you to call a Component Object library API function from your own code in an EXE. In either case, an API function is used to get the class factory pointer from your code to OLE 2's code.

DLL Server

The DLL exposure mechanism is the simplest, so let's start there. Every DLL server must export a function named *DllGetClassObject* with the following form:

```
HRESULT __export FAR PASCAL DllGetClassObject(REFCLSID rclsid,
    REFIID riid, LPVOID FAR *ppv);
```

The *__export* is a matter of convenience—if your compiler does not support *__export*, you can still list the function in the EXPORTS section of your DEF file. In addition, the macro STDAPI defined in COMPOBJ.H expands to *HRESULT __export FAR PASCAL* if you want to use it. In the sample code, you will see *EXPORT* instead of *__export*. *EXPORT* is a macro in BOOK1632.H (in the INC directory) that compiles to __export for Windows 3.1 and to nothing for Windows NT.

When a user calls *CoCreateInstance* or *CoGetClassObject* and passes CLSCTX_INPROC_SERVER, the Component Object library will look in the Registration Database for the *InprocServer* for the given CLSID, call *CoLoadLibrary* to get that server into memory, and then call *GetProcAddress* looking for *DllGetClassObject*. The Component Object library then calls *DllGetClassObject* with the CLSID and IID requested by the component user. Your export

then creates the appropriate class factory for the CLSID and returns the appropriate interface pointer for IID, which is usually *IClassFactory*. By calling this function in your DLL, the Component Object library obtains a pointer to your class factory object; essentially, *DllGetClassObject* is an API function you implement for OLE 2. This is exactly like exporting the WEP function from a DLL so that Windows can locate and call it.

NOTE: Because *DllGetClassObject* is passed a CLSID, a single DLL can provide different class factories for any number of different classes—that is, a single module can be the server for any number of object types. OLE2.DLL is an example of such a server; it provides most of the internally used object classes of OLE 2 from one DLL.

All implementations of *DllGetClassObject* should validate that it can support the requested CLSID as well as the requested interface for the class factory, which can be either *IUnknown* or *IClassFactory*. If both checks succeed, it then instantiates the class factory object (in this case, using *CKoalaClassFactory*). Remember that, as a function that creates a new interface pointer to an object, *DllGetClassObject* must use *AddRef* on the new object before returning, as shown in the following code:

```
if (!IsEqualCLSID(rclsid, CLSID_Koala))
    return ResultFromScode(CO_E_CLASSNOTREG);

//Check that we can provide the interface
if (!IsEqualIID(riid, IID_IUnknown)
    && !IsEqualIID(riid, IID_IClassFactory))
    return ResultFromScode(E_NOINTERFACE);

//Return our IClassFactory for Koala objects
*ppv=(LPVOID)new CKoalaClassFactory();

//Don't forget to AddRef the object through any interface we return
((LPUNKNOWN)*ppv)->AddRef();
```

Notice that this code, like all the sample code described in this book, creates all objects with an initial reference count of zero, thereby requiring a call to *AddRef* before returning an interface pointer to that object. This is simply the design approach taken in this book to reinforce the idea that a new interface pointer that you return to an external object user must have *AddRef* called through it so that the user can simply call *Release* when finished with the object to free it. You could, of course, set the reference count in your own objects to 1 in their constructors and avoid the explicit *AddRef* call shown here.

EXE Server

Exposing a class factory from an EXE is somewhat different because an EXE has a *WinMain*, a message loop, and a window that define its lifetime. The real difference between an EXE and a DLL in interacting with a Component Object library is that, with a EXE, instead of having the library call an exported function such as *DllGetClassObject*, you pass your class factory object (that is, an *IClassFactory* pointer) to the *CoRegisterClassObject* API function, but only under the appropriate circumstances.

The Component Object library informs an EXE that it is being used to service objects through the command-line flag *-Embedding* (which is left over from OLE 1). This flag is simply appended to the path entry for this local server in the Registration Database, so if you register your EXE with flags yourself, look for this at the end of the command line. Checking this flag is the first priority in EKOALA's initialization. If this flag is not present, the end user has attempted to run the application as a stand-alone from the shell. Because this application doesn't live for any purpose other than to service objects, it fails to load if *-Embedding* is not present.

The next few steps in *CAppVars::FInit* are the same as those required of any OLE 2 application: They use *CoBuildVersion* and *CoInitialize* because we are using Component Object library API functions. After such initialization, we create a window for this task, but the window remains hidden; in all cases in which *-Embedding* is on the command line, the server window should remain hidden until explicitly asked to show itself. For this demonstration, the EKOALA program has no need to ever show its window because it has no user interface.

If we get past the initialization stage, we must then create the class factory object and pass it to *CoRegisterClassObject* in the same way we are accustomed to calling the Windows API function *RegisterClass*. With *RegisterClass*, you create a WNDCLASS structure, fill in the *lpfnWndProc* field with a pointer to your window's message procedure, and pass a pointer to that WNDCLASS to *RegisterClass*. Your window procedure is not actually called until someone (you or a user) creates a window of your registered class. With *CoRegisterClassObject*, you create a class factory object with an *IClassFactory* interface and pass a pointer to that interface to *CoRegisterClassObject*, but the interface functions such as *CreateInstance* are not called until someone creates a component object of your class.[5]

5. In reality, calling *CoRegisterClassObject* immediately generates a number of calls to your *IClassFactory::AddRef* because the Component Object library is holding onto your *IClassFactory* pointer. Thus your object is called before it ever creates an object, unlike your window procedure.

Creating the class factory is simply a matter of allocating the object's data structure and function table, which is conveniently handled in C++ with the *new* operator, as follows:

```
//Return our IClassFactory for Koala objects
m_pIClassFactory=new CKoalaClassFactory();

if (NULL==m_pIClassFactory)
    return FALSE;

//Because we hold on to this, we should AddRef it
m_pIClassFactory->AddRef();
```

The additional *AddRef* ensures that the application controls the lifetime of the class factory because the *CKoalaClassFactory* constructor initializes its reference count to zero. Because the application makes the first *AddRef*, it will have to make the last *Release*, which allows the class factory to destroy itself.

After we have created the class factory, we must inform the Component Object library about it by using *CoRegisterClassObject* because we have yet to yield from this task in our message loop. The Component Object library does not have a chance to call us, as happens in a DLL.

```
hr=CoRegisterClassObject(CLSID_Koala, (LPUNKNOWN)m_pIClassFactory
    , CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE, &m_dwRegCO);

if (FAILED(hr))
    return FALSE;   //Registration failed.

[Class factory successfully registered]
```

CoRegisterClassObject takes the CLSID of the class factory we're providing, a pointer to the class factory, the context in which we're running (CLSCTX-_LOCAL_SERVER), a flag indicating how this class factory can be used, and a pointer to a DWORD in which *CoRegisterClassObject* returns a registration key that the object will need later, during shutdown.

NOTE: If your EXE is the server for multiple classes, you must call *CoRegisterClassObject* for each supported CLSID, just as you would call *RegisterClass* for each window class you support. The Component Object library will launch your object when any user requests any CLSID you support, but it cannot tell you through *WinMain* which CLSID that was. So you must register a class factory for each CLSID you placed in the Registration Database.

The fourth parameter to *CoRegisterClassObject* specifies how many objects can be created using this class factory: REGCLS_SINGLEUSE or

REGCLS_MULTIPLEUSE. If you specify single use, OLE will launch another instance of your application each time a user calls *CoGetClassObject*. If you specify multiple use, one instance of the application can service any number of objects. When you register a class with REGCLS_MULTIPLEUSE and CLSCTX_LOCAL_SERVER, the Component Object library also registers the class as CLSCTX_INPROC_SERVER. If you need to separately control whether the class factory is registered for local servers and in-process servers, use the flag REGCLS_MULTI_SEPARATE, which is available in OLE version 2.01 and later but not in the first release of OLE 2, version 2.00.

To demonstrate single-use vs. multi-use servers, run two instances of the OBJUSER program, choose Use EXE Object from the Koala Object menu of each instance, and use one of the Create commands from the menu. Now watch the modules that load and unload by using tools such as Heapwalker (in the Windows SDK) or WPS (in the OLE 2 SDK). Because EKOALA registers itself as multiple use, only one instance will be loaded to service both objects. Now change EKOALA so that it registers as single use and run two OBJUSERs again, using the same commands. This time two instances of EKOALA will run, each servicing only one object.

Also note that *CoRegisterClassObject* is not a function that can be called only from within an EXE. For all OLE cares, a DLL can call this function if it wants to expose a class factory outside of its implementation of *DllGetClassObject* or in lieu of *DllGetClassObject* altogether. The use flags should always be REGCLS_MULTIPLEUSE in such situations.

Provide an Unloading Mechanism

Because the mechanisms we use to expose a class factory from the two kinds of servers differ, the mechanisms for indicating when the server is no longer needed also differ. An unloading mechanism is not a consideration for normal Windows applications because they are almost always controlled by the user. OLE 2 allows DLLs and EXEs that serve objects to be controlled by another piece of component user code. Because the end user doesn't close applications, you must use a programmatic technique to accomplish the same end.

The bottom line is that a server is no longer needed when there are no lock counts from *IClassFactory::LockServer* and there are no objects currently being serviced. However, because the EXE server has a window, it must destroy its main window, cause a call to *PostQuitMessage*, exit the message loop, and quit the application. DLLs have no idea of how to "quit" (that is, there is no message loop to exit), so they mark themselves as "unloadable."

DLL Server

Again, let's start with the DLL because in this case the unload mechanism is trivial. As we have seen, the DLL server increments and decrements a global[6] lock count in *IClassFactory::LockServer* and increments the object count in *IClassFactory::CreateInstance*. When any Koala object is destroyed, we want to decrement the object count, a process that is handled in the *ObjectDestroyed* function we provided to the Koala object:

```
void FAR PASCAL ObjectDestroyed(void)
    {
    g_cObj--;
    return;
    }
```

The DLL never tells anyone to unload it; instead, the Component Object Model will periodically ask it "Can you unload now?" by calling an export *DllCanUnloadNow* using the following form:

```
STDAPI DllCanUnloadNow(void)
    {
    SCODE   sc;

    //Our answer is whether there are any object or locks
    sc=(0L==g_cObj && 0==g_cLock) ? S_OK : S_FALSE;
    return ResultFromScode(sc);
    }
```

The implementation shown here will answer "yes" when both object and lock counts are zero and "no" otherwise. If this function answers "yes," the libraries will internally call *CoFreeLibrary* to reverse the call the function made to *CoLoadLibrary* from within *CoGetClassObject*.

NOTE: The function that should call *DllCanUnloadNow* is *CoFreeUnusedLibraries*, which, as we've seen, is called periodically by an object user. However, the OLE 2 implementation of *CoFreeUnusedLibraries* does nothing, so you will never see a call to *DllCanUnloadNow*. However, *CoFreeUnusedLibraries* will be implemented in the near future, so implement *DllCanUnloadNow* as if it were always called anyway.

---

6. I confess! I used global variables! I normally try hard to avoid any use of global variables, as you probably do. In this case, however, having a few globals simplified both DLL and EXE implementation. You might also see me declare a global instance handle when appropriate because instance handles are really application-wide and might be needed deep in a long chain of function calls. In any case, global variables in this and other sample applications are prefixed with *g_* for clear identification. Please forgive my transgressions!

Also note that there has been no mention of class factory reference counts in any of this discussion because such reference counts are not used to keep the DLL in memory. Object user code wanting to hold a class factory must also call *LockServer* (as described earlier, in "Implementing a Component Object and Server"). Although a reference count could easily prevent a DLL from unloading, it's impossible to use this technique in an EXE, as the following section illustrates.

Congratulations! You're a proud parent! After implementing *DllCanUnloadNow*, you now have a complete DLL object server into which you can put more and more complex objects and interfaces, continuing to use the same mechanisms. The framework for DLL-based objects developed here will be used for more complex DLL objects later in this book. I certainly hope you will be able to use it for incredible objects of your own.

EXE Server

Instead of being asked when your object can be unloaded, as in the DLL case, an EXE server must initiate shutdown itself when it detects the following conditions: No objects are being serviced, and there is a zero lock count. This detection complicates use of EXE servers when we deal with compound documents because we throw in another condition regarding end-user control. (See the section titled "Call Initialization Functions at Startup and Shutdown" in Chapter 10.) When these two conditions are met, we need to start shutdown by posting a WM_CLOSE message to the main window. The two places where we must add a check are in *IClassFactory::LockServer* and in the *ObjectDestroyed* function, which the Koala object will call after it's freed:[7]

```
STDMETHODIMP CKoalaClassFactory::LockServer(BOOL fLock)
    {
    if (fLock)
        g_cLock++;
    else
        {
        g_cLock--;

        //No more objects and no locks, shut the app down.
        if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
            PostMessage(g_hWnd, WM_CLOSE, 0, 0L);
        }
```

(continued)

7. Another way to implement *LockServer(FALSE)* is to artificially increment the object count (g_cObj++) and call *ObjectDestroyed*, which decrements that artificial count and starts shutdown as appropriate. This approach centralizes the closure conditions in *ObjectDestroyed* and is used in samples in later chapters.

```
    return NOERROR;
    }

void FAR PASCAL ObjectDestroyed(void)
    {
    g_cObj--;

    //No more objects and no locks, shut the app down.
    if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
        PostMessage(g_hWnd, WM_CLOSE, 0, 0L);

    return;
    }
```

To facilitate message posting, I sinned again in EKOALA by storing its window handle as a global variable. This slight bit of what you might consider "cheating" works cleanly and easily without your having to pass the window handle around. Such a global variable guarantees that any code in this application could start a shutdown with the same mechanism. We might also use *PostAppMessage*, but that requires some changes to the application's message loop, which wouldn't be any cleaner.

By posting WM_CLOSE, we start shutdown of EKOALA exactly as if an end user had closed it from the system menu. In the process of shutting down, EKOALA destroys the main window (*DefWindowProc*'s handling of WM_CLOSE), exits *WinMain* (by calling *PostQuitMessage* in WM_DESTROY), and ends up in *CAppVars::~CAppVars*. This destructor first calls *CoRevokeClassObject*, which unregisters the class factory you passed to *CoRegisterClassObject* identified by the DWORD key that *CoRegisterClassObject* returned. If you registered multiple class factories for different CLSIDs, you must revoke each one here. *CoRevokeClassObject* will call *Release* for any reference count that the Component Object library was holding on the class factory. Furthermore, because we called *AddRef* ourselves before *CoRegisterClassObject* (remember that, long ago?), we must now match it with a *Release*. This will reduce the class factory's reference count to zero, causing it to free itself. Finally, because we called *CoInitialize*, we need to remember to call *CoUninitialize*.

CoRevokeClassObject is the reason why a Component User cannot use a class factory's reference count to keep a server—either a DLL or an EXE—in memory. If a positive reference count could keep the class factory in memory, we could not shut the application down until the reference count reached zero and the class factory was destroyed. The reference count will never reach zero unless we call *CoRevokeClassObject*, but we call *CoRevokeClassObject* only when we shut down after our window is gone, we've exited the message loop,

and we're on the nonstop express to oblivion. So we can't revoke until we're shutting down and we can't shut down until we revoke. Aaaugh! Fourth down and 100 yards to go...so we punt: Officially, a positive reference count on a class factory cannot be used to keep a server in memory, so a Component User must rely on *LockServer*, not the class factory reference count, to prevent shutdown. Our salvation is that this is one of the various special cases of reference counting in all of OLE 2.

Object Handlers

An object handler is a lightweight DLL server used to provide a partial implementation of a full EXE object, thereby reducing the need to launch the EXE to service that object. Because DLL objects generally load faster and need no parameter marshaling, use of handlers generally increases overall performance. Object handlers used in conjunction with a compound document object provide for data transfer and object rendering (directly to screen or printer) but not for editing, making handlers ideal in cases of licensing for redistribution with a document, in much the same way TrueType fonts in Windows 3.1 can be saved with a document, given the proper license. Chapter 11 will examine Object Handlers for specific use with compound documents.

If an object user calls *CoGetClassObject* with CLSCTX_INPROC_HANDLER ¦ CLSCTX_LOCAL_SERVER, the handler is loaded first and all calls to the object's interfaces are sent to the DLL. If a full DLL object exists as well, OLE will use that DLL first if the CLSCTX_INPROC_SERVER flag is specified.

When a handler discovers that it cannot provide the requested function to the caller, it can delegate that call to a full implementation of the object in an EXE server. However, the handler cannot simply call *CoCreateInstance* with CLSCTX_LOCAL_SERVER to launch the EXE and obtain a pointer to an object in that EXE. Instead the handler must instantiate what is called the *default handler* (OLE2.DLL) for the same object CLSID through the mechanism known as *aggregation*, as we'll see shortly. The handler then delegates the function call to the object in the default handler, which launches the EXE as appropriate. When the object in the handler wants to free itself, it also frees the default handler object, which, in turn, closes the EXE as necessary.

Of course, this is not without restrictions: The EXE object can support only standard interfaces with built-in marshaling support, and there is limited communication between the handler and the application. Furthermore, the handler must ensure that data is synchronized between itself and the application. Most often, however, a handler exists to provide speedy rendering of specific data formats and delegates requests for more esoteric formats to the application.

Finally, the only differences between DLL object handlers and DLL object servers are their intended use and expected performance. Technically and structurally, the handler is identical to a DLL server. All discussion in this chapter dealing with DLL servers applies equally to DLL handlers. The most significant differences are in how the two are used and how they should be designed, which is a topic for Chapter 11.

Cosmo's Polyline as a DLL Object

The Koala object that supports the *IPersist* interface is pretty boring and, well, useless. To demonstrate an object much more useful and exciting, the CHAP04\POLYLINE directory in the sample code contains an implementation of Cosmo's *CPolyline* class as a Windows Object in a DLL. The source code for POLYLINE.DLL (which compiles into POLY04.DLL) is a little too long to show here, however. This implementation shows that a much more complicated object, such as Polyline, can fit into exactly the same housing (DLL or EXE) as a simple object, such as Koala.

The member functions of the original *CPolyline* from Chapter 2 are converted into a custom interface, *IPolyline4*, defined in INC\IPOLY4.H and shown in Listing 4-6. (The *4* in these names stands for Chapter 4 because we'll be making modifications to the interface in later chapters.) Note that because all interface functions should return HRESULT whenever possible, some return values in the original *CPolyline* are converted into *out-parameters* in the interface. IPOLY4.H also defines the interface *IPolylineAdviseSink4*, through which a Cosmo document receives notifications from Polyline. This replaces the *CPolylineAdviseSink* class that Cosmo used before.

If you look in Polyline's sources, you'll notice that the core implementation of Polyline has not changed significantly from what was in Cosmo. In addition, the DLLPOLY.CPP file is only a slight modification of the DKOALA.CPP file: a few name changes, a class registered in *LibMain*, and a

IPOLY4.H

```
/*
 * Polyline Object Chapter 4
 *
 * Definition of an IPolyline interface for a Polyline object used
 * in the Cosmo implementation. This interface is custom and is
 * only supported from DLL-based objects.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _IPOLY4_H_
#define _IPOLY4_H_

//Versioning.
#define VERSIONMAJOR                 2
#define VERSIONMINOR                 0
#define VERSIONCURRENT               0x00020000

#define CPOLYLINEPOINTS              20

//Version 2 Polyline structure
typedef struct __far tagPOLYLINEDATA
    {
    WORD        wVerMaj;                //Major version number
    WORD        wVerMin;                //Minor version number
    WORD        cPoints;                //Number of points
    BOOL        fReserved;              //Previously fDrawEntire
    RECT        rc;                     //Rectangle of figure
    POINT       rgpt[CPOLYLINEPOINTS];  //Points on a 0-32767 grid

    //Version 2 additions
    COLORREF    rgbBackground;          //Background color
    COLORREF    rgbLine;                //Line color
    int         iLineStyle;             //Line style
    } POLYLINEDATA, *PPOLYLINEDATA, FAR *LPPOLYLINEDATA;

#define CBPOLYLINEDATA   sizeof(POLYLINEDATA)

//We use the OLE 2 macro to define a new interface
#undef  INTERFACE
#define INTERFACE IPolylineAdviseSink4
```

Listing 4-6. *(continued)*
IPolyline4 *and* IPolylineAdviseSink4 *custom interfaces.*

Listing 4-6. *continued*

```
/*
 * When someone initializes a polyline and is interested in receiving
 * notifications of events, they provide one of these objects.
 */

DECLARE_INTERFACE_(IPolylineAdviseSink4, IUnknown)
    {
    //IUnknown members
    STDMETHOD(QueryInterface) (THIS_ REFIID, LPLPVOID) PURE;
    STDMETHOD_(ULONG, AddRef)  (THIS) PURE;
    STDMETHOD_(ULONG, Release) (THIS) PURE;

    //Advise members.
    STDMETHOD_(void, OnPointChange)     (THIS) PURE;
    STDMETHOD_(void, OnSizeChange)      (THIS) PURE;
    STDMETHOD_(void, OnDataChange)      (THIS) PURE;
    STDMETHOD_(void, OnColorChange)     (THIS) PURE;
    STDMETHOD_(void, OnLineStyleChange) (THIS) PURE;
    };

typedef IPolylineAdviseSink4 FAR *LPPOLYLINEADVISESINK;

#undef  INTERFACE
#define INTERFACE IPolyline4

DECLARE_INTERFACE_(IPolyline4, IUnknown)
    {
    //IUnknown members:
    STDMETHOD(QueryInterface) (THIS_ REFIID, LPLPVOID) PURE;
    STDMETHOD_(ULONG, AddRef)  (THIS) PURE;
    STDMETHOD_(ULONG, Release) (THIS) PURE;

    //IPolyline members

    //File-related members:
    STDMETHOD(ReadFromFile) (THIS_ LPSTR) PURE;
    STDMETHOD(WriteToFile)  (THIS_ LPSTR) PURE;

    //Data transfer members:
    STDMETHOD(DataSet)      (THIS_ LPPOLYLINEDATA, BOOL, BOOL) PURE;
    STDMETHOD(DataGet)      (THIS_ LPPOLYLINEDATA) PURE;
    STDMETHOD(DataSetMem)   (THIS_ HGLOBAL, BOOL, BOOL, BOOL) PURE;
    STDMETHOD(DataGetMem)   (THIS_ HGLOBAL FAR *) PURE;
    STDMETHOD(RenderBitmap) (THIS_ HBITMAP FAR *) PURE;
    STDMETHOD(RenderMetafile) (THIS_ HMETAFILE FAR *) PURE;
    STDMETHOD(RenderMetafilePict) (THIS_ HGLOBAL FAR *) PURE;
```

(continued)

Listing 4-6. *continued*

```
    //Manipulation members:
    STDMETHOD(Init)    (THIS_ HWND, LPRECT, DWORD, UINT) PURE;
    STDMETHOD(New)     (THIS) PURE;
    STDMETHOD(Undo)    (THIS) PURE;
    STDMETHOD(Window)  (THIS_ HWND FAR *) PURE;

    STDMETHOD(SetAdvise) (THIS_ LPPOLYLINEADVISESINK) PURE;
    STDMETHOD(GetAdvise) (THIS_ LPPOLYLINEADVISESINK FAR *) PURE;

    STDMETHOD(RectGet) (THIS_ LPRECT) PURE;
    STDMETHOD(SizeGet) (THIS_ LPRECT) PURE;
    STDMETHOD(RectSet) (THIS_ LPRECT, BOOL) PURE;
    STDMETHOD(SizeSet) (THIS_ LPRECT, BOOL) PURE;

    STDMETHOD(ColorSet) (THIS_ UINT, COLORREF
        , COLORREF FAR *) PURE;
    STDMETHOD(ColorGet) (THIS_ UINT, COLORREF FAR *) PURE;

    STDMETHOD(LineStyleSet) (THIS_ UINT, UINT FAR *) PURE;
    STDMETHOD(LineStyleGet) (THIS_ UINT FAR *) PURE;
    };

typedef IPolyline4 FAR *LPPOLYLINE;

//Error values for data transfer functions
#define POLYLINE_E_INVALIDPOINTER    \
    MAKE_SCODE(SEVERITY_ERROR, FACILITY_ITF, 1)
#define POLYLINE_E_READFAILURE       \
    MAKE_SCODE(SEVERITY_ERROR, FACILITY_ITF, 2)
#define POLYLINE_E_WRITEFAILURE      \
    MAKE_SCODE(SEVERITY_ERROR, FACILITY_ITF, 3)

//Color indices for color member functions
#define POLYLINECOLOR_BACKGROUND    0
#define POLYLINECOLOR_LINE          1

#endif //_IPOLY4_H_
```

DLL instance handle passed to the *CPolyline* constructor. Oh yes, *CPolyline* still exists, but it is now more like the *CKoala* object in the previous examples. The member functions of the *CPolyline* of Chapter 2 have been moved to the interface implementation *CImpIPolyline*.

As we move forward in this book, we'll incrementally replace specific members of *IPolyline4* with those of another interface. In the next chapter, for example, we'll remove the two file-related functions from *IPolyline5*, replacing them with an *IPersistStorage* interface on the object. In Chapter 6, we'll replace the data transfer and graphics rendering functions in *IPolyline6* with the *IDataObject* interface. Beyond that, we'll add compound document features, including in-place activation, to Polyline. However, all these additions come in the form of other interfaces, which will not interfere with Polyline's operation as a component object.

The version of Cosmo that uses the component Polyline object, Component Cosmo, is provided in CHAP04\COCOSMO and requires only a few modifications. When you run Component Cosmo, however, you will notice absolutely no changes in the user interface or in any behavior. Component Cosmo merely changed from being the user of a local C++ object, *CPolyline*, to being a Component User of the Polyline Windows object through the *IPolyline4* interface.

Object Reusability

So what about inheritance?

Windows Objects themselves and the classes they identify through CLSIDs have no notion of *implementation inheritance* whatsoever. One Windows Object does not inherit the implementation of another Windows Object. But Windows Objects are still *reusable* through two mechanisms, *containment* and *aggregation*. These mechanisms have several significant benefits over inheritance, which is why the Component Object Model has significant benefits over models that rely heavily on inheritance.

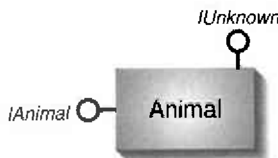
In the Component Object Model, inheritance is simply considered a tool that is useful for implementing classes in C++, as well as for defining interfaces. In your implementation of an object, either you can use multiple inheritance from all the interfaces you support, or you can contain implementations of each interface the object supports, in which each interface implementation inherits a single interface. That's really what an interface is for: to help the *implementor* of an object but not the object *user.* Inheritance greatly enhances programmer productivity, but does the user really care how the object was implemented? The answer is definitely "NO"—object users are *supposed* to be entirely ignorant of the object's implementation, especially when the implementation exists in other pieces of code that you did not implement or for which you don't have the source, such as Windows itself.

The single most significant problem of inheritance is that two unrelated pieces of code work on the instance of an object. If I have a base class B and a derived class D, which inherits from B, an instantiation of class D is only one data structure in memory. If B contains virtual functions that D does not override or if the implementation of D explicitly calls a member function in B (that is, using *B::<member function>*), we again have two pieces of code working on the same memory. But class B does not know the expected behavior of class D, so how on earth can D force the correct behavior of its objects? The answer is that D must know what B is going to do on that object so that it knows when to override a virtual function in B and when exactly to explicitly call B's functions. This is exactly the same problem as trying to figure out when to call *DefWindowProc* for any given message: We've merely replaced the word *message* with *virtual function*. In any case, because B cannot know about the implementation of D, D must know about the implementation of B, which causes D to violate its status as a user of B.

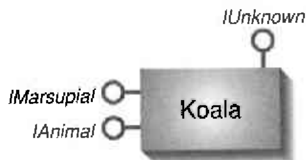
Systems built on inheritance have the key problem that they must ship all their source code in order to be usable. Take a look at application frameworks such as the Microsoft Foundation Classes: Source code is shipped with the product, so you know how to inherit from any given class and can duplicate behavior as appropriate. Sure, inheritance works well in building large, complex systems because it's a much better way to manage source code than creating a large stockpile of sample source files. It certainly works well when you control and have access to all the source code for all classes. It certainly helped me to develop the sample applications in this book. But it does not work for reusing objects implemented in the operating system itself, for which either source code is not available or you did not originally implement the object yourself.

The Component Object Model avoids all these problems but retains reusability through its mechanisms of containment and aggregation, which we are finally in a position to explore in detail. Both mechanisms achieve reusability literally by using, instead of inheriting, the implementation of another object. The object we're using remains entirely self-contained and operates on its own instance of data. Our own object, which is called the *aggregate,* works on its own instance of data and calls the other object as necessary to perform specific functions in which we can pass it the data on which to operate.

Let's say I have an object named Animal that knows only of itself and exists as an atomic entity (like the Koala object). I can illustrate this object as a block with circular jacks for each interface: *IUnknown* and *IAnimal* (with members such as *Eat, Sleep,* and *Procreate*). Again, by convention, *IUnknown* is always shown on top, with all other interfaces shown to the side, as follows:



A user of this object with a pointer to either interface can use *QueryInterface* to get a pointer to the other. The implementation of *IAnimal* knows about the object's *IUnknown* and vice versa. Now I want to create a more complicated Koala object that will expose interfaces *IUnknown*, *IAnimal*, and *IMarsupial* (maybe with members such as *CarryYoungInPouch* and *LiveInAustralia*) with a more complicated picture:[8]



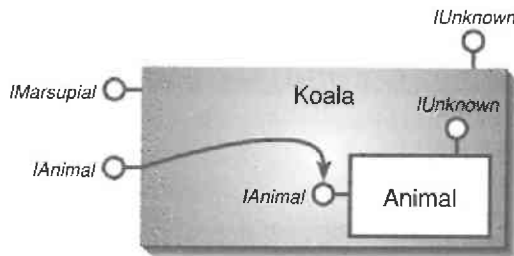
When I implement Koala, I know that Animal exists and I want to reuse Animal's implementation. I can use Animal and its implementation of *IAnimal* in the following two ways, neither of which changes how the external world sees Koala:

- Containment: Koala completely contains an Animal object and implements its own version of *IAnimal* to expose externally. This makes Koala a simple user of Animal, and Animal need not care. Koala never calls *IAnimal::QueryInterface.*
- Aggregation: Koala exposes Animal's *IAnimal* interface directly as Koala's *IAnimal.* This requires that Animal know that its interface is exposed for something other than itself, such that *QueryInterface*, *AddRef*, and *Release* behave as a user expects.

8. Instinct tells you that *IMarsupial* should inherit from *IAnimal* because a marsupial is just another kind of animal. The Windows Object notion of interfaces, however, means that through a pointer to *IMarsupial*, you deal with the object as a marsupial but not generally as anything else. If you want to treat it the same as any other animal, call *IMarsupial::QueryInterface(IID_IAnimal)* for the appropriate interface. As a real life example, consider Compound Document objects that are all treated through *IOleObject*, regardless of whether they are linked or embedded. A linked object can be viewed as a further refinement of an embedded object, so you might expect that an interface such as *IOleLink* for linked objects would inherit from *IOleObject*. But it doesn't. You use *QueryInterface* through *IOleObject* for *IOleLink*. *QueryInterface* is the mechanism for getting at more functions on the same object.

Case 1: Object Containment

Complete containment of Animal is necessary when I need to change some aspect of my implementation of *IAnimal*. Because all external calls to that interface will enter Koala first, Koala can override specific functions or simply pass that call to Animal's implementation. The internal structure of Koala will appear something like this:



In this case, Animal always operates on its own data unless Koala explicitly passes other data to it (which is also true in aggregation). In other words, by default the two objects work on different data, and only by conscious design of an interface would the two be able to communicate. This is much different from inheritance, in which working on the same data is the default and it takes conscious effort to create separate data instances.

To build this sort of structure, Koala calls *CoCreateInstance* on *CLSID_Animal* when Koala itself is created, passing a NULL for *pUnkOuter* and asking for *IID_IAnimal*. Koala maintains this *IAnimal* pointer until the Koala object is destroyed, at which time Koala calls *IAnimal::Release* to free the Animal object. Whenever Koala's *IAnimal* implementation wants to reuse Animal's *IAnimal* implementation, Koala simply calls the appropriate member function on Animal.

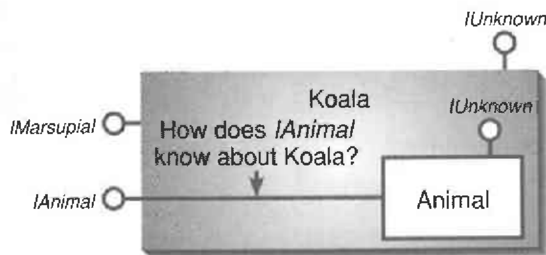
Reusing an object through containment is much like using a Windows list box to manage a list of information. For example, the Patron program in Chapter 2 maintains a list box in such a way that each item is a pointer to a page in the document. The list box provides all the memory management to maintain the list, removing that burden from the application. But Patron never makes the list box visible, so nothing outside of Patron knows that it's using a list box in this manner. Containment is the same, in that the aggregate object uses specific services of the contained object without ever showing the outside world that it is using the contained object in this capacity.

This technique is the simplest way to reuse another's implementation of an interface. However, you do not always care to override *any* functions in such an interface, wanting only to pass every call through to the object you're

using. You could, of course, implement stubs for every *IAnimal* function that only calls the contained object, but you would rather simply expose that interface directly and eliminate any need for such stubs. That technique is aggregation.

Case 2: Object Aggregation

Aggregation on Animal is useful when Koala does not want to change any aspect of how it appears through the *IAnimal* interface—that is, Koala has no need simply to implement a bunch of Animal stubs that only delegate to a contained Animal object. Therefore, Koala wants to expose the *IAnimal* interface of the Animal object directly, turning it into Koala's *IAnimal*. This yields an internal structure like the following:

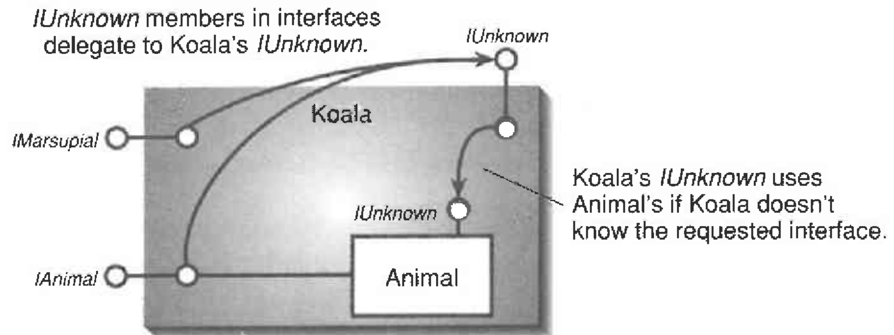


Here's the problem. Because Animal's *IAnimal* is exposed directly, users of Koala will expect that *IAnimal::QueryInterface(IID_IMarsupial)* will return a pointer to Koala's *IMarsupial*. But Animal was not written to know anything about *IMarsupial*, let alone know anything about the Koala object. How can it know the identity of the outer object and its interfaces?

The answer is that when Koala creates Animal, Koala passes its *IUnknown* pointer to the Animal class factory *CreateInstance* as the *pUnkOuter* parameter. (Note that Animal holds onto this pointer but does not—repeat, not—call *AddRef* through it.) In this fashion, Koala identifies itself as the *controlling unknown* of the aggregate. Furthermore, Koala must always ask for Animal's *IUnknown* when creating Animal as part of an aggregate. Note that an object might not support aggregation, in which case it fails *CreateInstance* when a non-NULL *pUnkOuter* is specified.

This sets up a contract between the aggregate object (Koala) and the aggregatee (Animal) in such a way that Animal *must* implement an instance of *IUnknown* that is separate from all other interfaces. This *IUnknown* can return interface pointers to all other interfaces on Animal, and so Koala can ask Animal's *IUnknown* for any of Animal's interfaces and expose those interfaces as if they belonged to Koala. Now the *QueryInterface, AddRef,* and

Release functions in any of Animal's interfaces—besides Animal's separate *IUnknown*—don't do anything to Animal. Instead these functions call the same functions in Koala's controlling unknown because these functions affect the object as a whole *as seen from the outside.* From the outside, Animal's interfaces appear as if they were interfaces on Koala, and so they must act like interfaces on Koala. This means that *AddRef* and *Release* affect Koala's reference count and that *QueryInterface* can return pointers to any interface exposed from Koala.



The *pUnkOuter* parameter passed to Animal's *IClassFactory::CreateInstance* must be available to all interfaces in Animal except for Animal's *IUnknown* itself. Typically this means that Animal stores *pUnkOuter* in its object structure:

```
STDMETHODIMP CAnimalClassFactory(LPUNKNOWN pUnkOuter, ...)
    {
    LPUNKNOWN   pObj;
    ...
    pObj=new CAnimal(pUnkOuter, ...);    //Create the object
    ...
    pObj->FInit();                       //Initialize object
    ...
    }

CAnimal::CAnimal(LPUNKNOWN pUnkOuter, ...)
    {
    ...
    m_pUnkOuter=pUnkOuter;   //Save the controlling unknown
    ...
    }
```

Note that when CAnimal saves *pUnkOuter* in this example, it does not call *AddRef* through that pointer. This is because Animal's lifetime is entirely defined by Koala's lifetime, so the *AddRef* is unnecessary and dangerous: If Animal held a reference to Koala, Koala could not free itself until Animal released that reference. But Animal will not release that reference until it frees itself, and that will not happen until Koala releases its reference on Animal, which only happens if Koala is freeing itself. If we had to choose between death and spinning around in this endless reference loop, we'd keep circling until death became the favorable alternative. To avoid such problems, the aggregatee (Animal, in this case) is specifically required to *not* call *AddRef* through *pUnkOuter*.

Now Animal's *IUnknown* must not delegate to the controlling unknown; instead it must return only those interfaces known to Animal, and it must only affect Animal's reference count. This *IUnknown* implementation essentially controls Animal's lifetime:

```
STDMETHODIMP CAnimal::QueryInterface(REFIID riid, LPLPVOID ppv)
    {
    *ppv=NULL;

    if (IsEqualIID(riid, IID_IUnknown))
        *ppv=(LPVOID)(LPUNKNOWN)this;

    if (IsEqualIID(riid, IID_IAnimal))
        *ppv=(LPVOID)m_pIAnimal;

    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    return ResultFromScode(E_NOINTERFACE);
    }
```

In this code, *m_pIAnimal* is a pointer to Animal's implementation of the *IAnimal* interface, which it creates in *CAnimal::FInit*:

```
BOOL CAnimal::FInit(void)
    {
    LPUNKNOWN       pIUnknown=(LPUNKNOWN)this;

    if (NULL!=m_pUnkOuter)
        pIUnknown=m_pUnkOuter;
```

(continued)

```
    //Allocate contained interfaces.
    m_pIAnimal=new CImpIAnimal(this, pIUnknown);

    return (NULL!=m_pIAnimal);
    }
```

Animal here plays a nasty trick on its interface implementation, which, as you may recall, must delegate to the controlling unknown if Animal is being aggregated but must affect Animal outside of aggregation. To handle this, Animal always passes some *IUnknown* to the interface implementation. Under aggregation, it's the controlling unknown. Without aggregation, Animal passes its own *IUnknown* as the controlling unknown to the interface. The interface, in turn, blindly delegates all *IUnknown* calls to whatever controlling unknown it was given, as follows:

```
CImpIAnimal::CImpIAnimal(LPVOID pObj, LPUNKNOWN pUnkOuter)
    {
    ...
    m_punkOuter=pUnkOuter;
    ...
    }

STDMETHODIMP CImpIAnimal::QueryInterface(REFIID riid, LPLPVOID ppv)
    {
    return m_pUnkOuter->QueryInterface(riid, ppv);
    }

STDMETHODIMP_(ULONG) CImpIAnimal::AddRef(void)
    {
    return m_pUnkOuter->AddRef();
    }

STDMETHODIMP_(ULONG) CImpIAnimal::Release(void)
    {
    return m_pUnkOuter->Release();
    }
```

So let's say there is no aggregation; the *m_pUnkOuter* to which *CImplAnimal* delegates is the *IUnknown* implemented in *CAnimal.* This *IUnknown* implementation will return pointers for *IUnknown* and *IAnimal,* as shown before.

If there is an aggregate object, *m_pUnkOuter* points to the controlling unknown, so the calls to this *IUnknown* from the *IAnimal* interface bypass Animal's *IUnknown* and end up in Koala's *IUnknown.* In this controlling unknown, Koala handles *AddRef* and *Release* calls as if they were made through one of its own interfaces. As for *QueryInterface,* the controlling unknown can handle it in one of three ways:

- If the requested interface is implemented by the aggregate object, return a pointer to that interface directly.
- If the requested interface is implemented in the contained object and exposed as one of the aggregate's interfaces, delegate the *QueryInterface* to the contain object's *IUnknown*.
- If the aggregate object does not recognize the reqeusted interface it may either blindly delegate the request to the contained object or fail, depending on the goals of the aggregate object.

The *QueryInterface* of an aggregate's controlling unknown, such as the one Koala would implement, would therefore appear follows:

```
STDMETHODIMP CKoala::QueryInterface(REFIID riid, LPLPVOIDppv)
    {
    *ppv=NULL;

    if (IsEqualIID(riid, IID_IUnknown))
        *ppv=(LPVOID)this;

    if (IsEqualIID(riid, IID_IMarsupial))
        *ppv=(LPVOID)m_pIMarsupial;

    if (IsEqualIID(riid, IID_IAnimal))
        return m_pIUnknownAnimal->QueryInterface(riid, ppv);

    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    return ResultFromScode(E_NOINTERFACE);
    }
```

Here, *m_pIUnknownAnimal* is the *IUnknown* pointer we requested when creating the Animal object. The *IUnknown* implementation on an object such as *CAnimal* in this example is always the last stop for a *QueryInterface*, an *AddRef*, or a *Release* call. Animal's *IUnknown* never worries about aggregation in and of itself because it has to be the controlling unknown for its own object. Only if the Animal object itself contained more primitive objects would it do any further delegation, but in no way will it pass any request to a higher unknown.

If the requested interface was not *IUnknown*, *IMarsupial*, or *IAnimal*, what should Koala do? Koala has two choices, depending on how it's using the Animal object. First, Koala might be using Animal as what I call a "helper" object. A helper object is used to provide very specific services to higher aggregate objects where the aggregate objects expose only very specific interfaces from the helper object iself. The preceding code uses the Animal object in this fashion. This is similar to using a hidden list box control in an application to perform list management. You never show the list box, but you are using it as a helper for your implementation.

The second way in which Koala might use Animal is where Koala's *IUnknown* would itself delegate *QueryInterface* calls for any unrecognized interface to Animal's IUnknown. This approach would replace the *return ResultFromScode(...)* call in the preceding code with *m_pIUnknownAnimal->QueryInterface(...))*. By doing so, Koala is not aware of all the interfaces that it might expose as its own. For example, if Animal implemented an additional interface called *IPrimate*, Koala would look like both a marsupial and a primate at the same time. This type of aggregation is useful only when Koala is essentially subclassing the Animal object by adding an interface of its own. This is exactly like subclassing a Windows edit control in such a way that your subclass procedure changes the behavior of the edit control for a few specific messages but blindly passes all other messages to the original message procedure of the control. The aggregate object in this case is only a thin shell around the contained object.

To wrap up our discussion about aggregation, I want to mention one more point. In its role as the aggregate object, Koala might want to cache specific interface pointers it obtains by calling Animal's *IUnknown::QueryInterface*. If it does so, it must do something strange: It must immediately call *Release* through the pointer as follows, where *m_pIUnknownAnimal* is the Animal's *IUnknown* and *m_pIAnimal* is a variable in the Koala object:

```
HRESULT   hr;

[Code that created m_pIUnknownAnimal]

//Cache a pointer to Animal's IAnimal
m_pIAnimal=NULL;

hr=m_pIUnknownAnimal->QueryInterface(IID_IAnimal
    , (LPLPVOID)&m_pIAnimal);

if (SUCCEEDED(hr))
    m_pIAnimal->Release();
```

After executing this code, *m_pIAnimal* will be either NULL (in which case, we could not cache the pointer) or non-NULL. But we called *Release* already, right? Doesn't that invalidate the pointer? Actually, it doesn't because we are still holding onto *m_pIUnknownAnimal,* and so the object itself is still valid, which means that all its interface pointers, including the one we just called *Release* through, also remain valid. Although this seems strange, think about the consequences. Animal's *IUnknown::QueryInterface* will call *IAnimal::AddRef* before returning the pointer to Koala. But *IAnimal* has Koala's *IUnknown* as the controlling unknown, so the *AddRef* call will increment Koala's reference count, which will not be decremented until Koala calls *m_pIAnimal->Release.* If Koala did not make the call here, it would be able to make the call only when it was freeing itself, but it could never get there because of this extra reference count. So, once again, to avoid the problems of circular references, the aggregate object must call *Release* through any pointers obtained from the contained object's *IUnknown.*

Summary

A first requirement of all OLE 2 applications is that they use a message queue of size 96 (under Windows 3.1) and that they provide for initializing the OLE 2 libraries if, in fact, the application can run with the version of those libraries that currently exists on the machine. Checking versions is accomplished through the *CoBuildVersion* and *OleBuildVersion* functions, whereas initialization occurs through *CoInitialize* and *OleInitialize.* On shutdown, an application must also call *CoUninitialize* or *OleUninitialize* to reverse the corresponding ...*Initialize* call. These requirements are presented in this chapter because all later samples must comply with them.

Part of library initialization involves defining a task allocator object, one that implements the *IMalloc* interface, which is used for all task memory allocations. An OLE 2 application can either implement its own or use an OLE 2-provided allocator that works on the technique of multiple local heaps. OLE 2 always implements a similar shared allocator that can provide memory shareable between applications. Although applications can change the task allocator, they cannot change the shared allocator. While the application is running, any other piece of code (such as the OLE 2 libraries) can call *CoGetMalloc* to obtain a pointer to either the task or shared allocator objects.

The Ultimate Question presented in Chapter 3 asked how you obtain a pointer, given knowledge and the identification of a specific Windows Object. This chapter deals with the specific case in which you identify a Windows Object, given a CLSID, and use the function *CoCreateInstance* to instantiate an

object of that class. Such an object is called a *component object*, and the application using it is called a *component user*. *CoCreateInstance* internally uses *CoGetClassObject*, which obtains a class factory object (*IClassFactory*) for the CLSID and calls *IClassFactory::CreateInstance* to perform the actual instantiation. What you do with the object once you have an interface pointer to it is your own business, although there are a few considerations when you release the object, such as calling *CoFreeUnusedLibraries* to purge unused DLLs from memory.

Implementations of component objects that can be loaded and called by functions such as *CoCreateInstance* and *CoGetClassObject* have different structural requirements, depending on whether the object lives in a DLL or an EXE. A DLL exports a function named *DllGetClassObject*, which provides the API function through which *CoGetClassObject* obtains a pointer to the DLL's class factory for a given CLSID. EXEs, on the other hand, must pass a pointer to their class factory to the *CoRegisterClassObject* function for each supported CLSID. The two module types also differ in their shutdown conditions. Whereas the Component Object library asks a DLL whether that DLL can be unloaded, an EXE must initiate its own shutdown when the proper conditions are met—that is, it must destroy its main window and exit its message loop. As examples, this chapter implements an object named Koala that supports the *IPersist* interface in both a DLL and an EXE and then separates the Polyline object of the sample Cosmo application into a component object and shows a modification of Cosmo, called Component Cosmo, which uses the component Polyline object.

Object reusability in OLE 2 is achieved through mechanisms called *containment* and *aggregation*, not through inheritance. The inheritance mechanism works well for source code management but generally requires that you have the source code available for any classes from which you inherit. Because of source code availability and a host of other problems, OLE 2 works on mechanisms other than inheritance that provide the same reusability of code, but it avoids the problems with traditional techniques. There is, however, some impact on the implementation of an object that wants to allow itself to be reusable via containment and aggregation.

OBJECT-ORIENTED SYSTEM FEATURES: FILES AND DATA TRANSFER