Davis Chapman

SAMS

Teach Yourself Visual C++® 6 in 21 Days

SAMS
A Division of Macmillan Computer Publishing
201 West 103rd St., Indianapolis, Indiana, 46290 USA

Sams Teach Yourself Visual C++® 6 in 21 Days

Copyright © 1998 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-31240-9

Library of Congress Catalog Card Number: 98-84508

Printed in the United States of America

First Printing: August, 1998

01 00 99 98 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Visual C++ is a registered trademark of Microsoft Corporation.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Executive Editor
Brad Jones

Acquisitions Editor
Kelly Marshall

Development Editor
Matt Purcell

Managing Editor
Jodi Jensen

Project Editor
Dana Rhodes Lesh

Copy Editor
Kris Simmons

Indexer
Erika Millen

Technical Editor
Larry Richardson

Production
Marcia Deboy
Michael Dietsch
Jennifer Earhart
Cynthia Fields
Susan Geiselman

Overview

	Introduction	1
Week 1 at a Glance		**5**
Day 1	The Visual C++ Development Environment—Building Your First Visual C++ Application	7
2	Using Controls in Your Application	25
3	Allowing User Interaction—Integrating the Mouse and Keyboard in Your Application	47
4	Working with Timers	67
5	Getting User Feedback—Adding Dialog Boxes to Your Application	83
6	Creating Menus for Your Application	105
7	Working with Text and Fonts	121
Week 1 in Review		**143**
Week 2 at a Glance		**147**
Day 8	Adding Flash—Incorporating Graphics, Drawing, and Bitmaps	149
9	Adding ActiveX Controls to Your Application	179
10	Creating Single Document Interface Applications	199
11	Creating Multiple Document Interface Applications	229
12	Adding Toolbars and Status Bars	243
13	Saving and Restoring Work—File Access	279
14	Retrieving Data from an ODBC Database	315
Week 2 in Review		**337**
Week 3 at a Glance		**341**
Day 15	Updating and Adding Database Records Through ADO	343
16	Creating Your Own Classes and Modules	379
17	Sharing Your Functionality with Other Applications—Creating DLLs	405

18	Doing Multiple Tasks at One Time—Multitasking	429
19	Building Your Own Widgets—Creating ActiveX Controls	473
20	Internet Applications and Network Communications	495
21	Adding Web Browsing Functionality to Your Applications	521

WEEK 3 IN REVIEW **537**

APPENDIXES **541**

A	C++ Review	541
B	Answers	579
C	Printing and Print Previewing	637
D	Understanding and Exception Handling	661
E	Using the Debugger and Profiler	677
F	Using MFC's Helper Classes	699
	Index	731

Contents

INTRODUCTION 1

WEEK 1 AT A GLANCE 5

DAY 1 THE VISUAL C++ DEVELOPMENT ENVIRONMENT—BUILDING YOUR FIRST VISUAL C++ APPLICATION 7

The Visual C++ Development Environment 8
- The Output Pane 9
- The Editor Area 9
- Menu Bars 10
- Rearranging the Developer Studio Environment 10

Starting Your First Project 11
- Creating the Project Workspace 11
- Using the Application Wizard to Create the Application Shell 12

Designing Your Application Window 15
Adding Code to Your Application 17
Finishing Touches 20
- Creating the Dialog Box Icon 20
- Adding Maximize and Minimize Buttons 21

Summary 22
Q&A 22
Workshop 23
- Quiz 23
- Exercise 23

DAY 2 USING CONTROLS IN YOUR APPLICATION 25

The Basic Windows Controls 26
- The Static Text Control 26
- The Edit Box Control 26
- The Command Button Control 27
- The Check Box Control 27
- The Radio Button Control 27
- The Drop-Down List Box Control 27

Adding Controls to Your Window 27
- Creating the Application Shell and Dialog Layout 28

Specifying the Control Tab Order 30
Attaching Variables to Your Controls 32
Attaching Functionality to the Controls 33
Closing the Application 36
Showing the User's Message 37
Clearing the User's Message 38
Disabling and Hiding the Message Controls 39
Running Another Application 42
Summary 44
Q&A 44
Workshop 45
Quiz 45
Exercises 45

DAY 3 ALLOWING USER INTERACTION—INTEGRATING THE MOUSE AND KEYBOARD IN YOUR APPLICATION 47
Understanding Mouse Events 48
Drawing with the Mouse 49
Improving the Drawing Program 53
Adding the Finishing Touches 55
Capturing Keyboard Events 56
Changing the Drawing Cursor 57
Making the Change Stick 60
Summary 64
Q&A 64
Workshop 65
Quiz 65
Exercises 65

DAY 4 WORKING WITH TIMERS 67
Understanding Windows Timers 68
Placing a Clock on Your Application 68
Creating the Project and Application 68
Adding the Timer IDs 70
Starting the Clock Timer 71
Handling the Clock Timer Event 72
Adding a Second Timer to Your Application 74
Adding the Application Variables 74
Starting and Stopping the Counting Timer 75
Enabling the Stop Button 79
Summary 81

Q&A 81
Workshop 82
Quiz 82
Exercise 82

DAY 5 GETTING USER FEEDBACK—ADDING DIALOG BOXES TO YOUR APPLICATION 83

Using Pre-existing (or System) Dialog Windows 84
Using Message Boxes 84
Using Common Dialogs 90
Creating Your Own Dialog Windows 93
Creating the Dialog Window 93
Using the Dialog in Your Application 96
Summary 101
Q&A 101
Workshop 103
Quiz 103
Exercises 104

DAY 6 CREATING MENUS FOR YOUR APPLICATION 105

Menus 106
Menu Styles 106
Keyboard Shortcut–Enabling Menus 106
Menu Standards and Conventions 107
Designing Menus 108
Creating a Menu 108
Creating the Application 108
Adding and Customizing a Menu 109
Attaching the Menu to Your Dialog Window 112
Attaching Functionality to Menu Entries 112
Creating Pop-Up Menus 116
Creating a Menu with Accelerators 118
Summary 119
Q&A 119
Workshop 120
Quiz 120
Exercises 120

DAY 7 WORKING WITH TEXT AND FONTS 121

Finding and Using Fonts 122
Listing the Available Fonts 122
Using a Font 125

Using Fonts 129
Creating the Application Shell 129
Building a List of Fonts....131
Setting the Font Sample Text 135
Selecting a Font to Display 136
Summary....139
Q&A 139
Workshop 140
Quiz 140
Exercises 140

WEEK 1 IN REVIEW 143

WEEK 2 AT A GLANCE 147

DAY 8 ADDING FLASH—INCORPORATING GRAPHICS, DRAWING, AND BITMAPS 149

Understanding the Graphics Device Interface 150
Device Contexts....150
Mapping Modes and Coordinate Systems....156
Creating a Graphics Application....157
Generating the Application Shell 157
Adding the Graphics Capabilities 163
Summary....177
Q&A 177
Workshop 177
Quiz 178
Exercises 178

DAY 9 ADDING ACTIVEX CONTROLS TO YOUR APPLICATION 179

What Is an ActiveX Control? 180
ActiveX and the `IDispatch` Interface 180
ActiveX Containers and Servers 182
Adding an ActiveX Control to Your Project 183
Registering the Control 183
Adding the Control to Your Dialog....186
Using an ActiveX Control in Your Application 187
Interacting with the Control 187
Responding to Control Events....193
Summary....196
Q&A 196
Workshop 197
Quiz 197
Exercise 197

DAY 10 CREATING SINGLE DOCUMENT INTERFACE APPLICATIONS 199

The Document/View Architecture 200
Creating an SDI Application....202
Building the Application Shell 202
Creating a Line Class 203
Implementing the Document Functionality....205
Showing the User 208
Saving and Loading the Drawing....213
Deleting the Current Drawing....214
Saving and Restoring the Drawing 215
Interacting with the Menu....218
Adding Color to the `CLine` Class 218
Adding Color to the Document....219
Modifying the Menu 222
Summary....226
Q&A 227
Workshop 228
Quiz 228
Exercise 228

DAY 11 CREATING MULTIPLE DOCUMENT INTERFACE APPLICATIONS 229

What Is an MDI Application?....229
Creating an MDI Drawing Program....231
Building the Application Shell 231
Building the Drawing Functionality 232
Adding Menu Handling Functionality 233
Adding a Context Menu 236
Summary....239
Q&A 239
Workshop 240
Quiz 240
Exercise 241

DAY 12 ADDING TOOLBARS AND STATUS BARS 243

Toolbars, Status Bars, and Menus 244
Designing a Toolbar 245
Creating a New Toolbar 246
Attaching the Toolbar to the Application Frame....247
Controlling the Toolbar Visibility 254
Adding a Combo Box to a Toolbar 257
Editing the Project Resources 257
Creating the Toolbar Combo Box 260

Handling the Toolbar Combo Box Events 267
Updating the Toolbar Combo Box 268
Adding a New Status Bar Element....271
Adding a New Status Bar Pane....271
Setting a Status Bar Pane Text 273
Summary....275
Q&A 276
Workshop 277
Quiz 277
Exercises 277

DAY 13 SAVING AND RESTORING WORK—FILE ACCESS 279

Serialization 279
The `CArchive` and `CFile` Classes....280
The Serialize Function....281
Making Objects Serializable 281
Implementing a Serializable Class 283
Creating a Serialized Application 284
Creating a Serializable Class....288
Building Support in the Document Class 294
Adding Navigating and Editing Support in the View Class 305
Summary....311
Q&A 311
Workshop 313
Quiz 313
Exercise 313

DAY 14 RETRIEVING DATA FROM AN ODBC DATABASE 315

Database Access and ODBC....316
The Open Database Connector (ODBC) Interface 316
The `CRecordset` Class....317
Creating a Database Application Using ODBC 322
Preparing the Database 322
Creating the Application Shell 324
Designing the Main Form 325
Adding New Records 331
Deleting Records....334
Summary....335
Q&A 335
Workshop 336
Quiz 336
Exercise 336

WEEK 2 IN REVIEW 337

WEEK 3 AT A GLANCE 341

DAY 15 UPDATING AND ADDING DATABASE RECORDS THROUGH ADO 343

What Is ADO?....344
ADO Objects....345
Using the ADO ActiveX Control....346
Importing the ADO DLL....349
Connecting to a Database....350
Executing Commands and Retrieving Data....351
Navigating the `Recordset`....352
Accessing Field Values....353
Updating Records....356
Adding and Deleting....356
Closing the `Recordset` and `Connection` Objects....357
Building a Database Application Using ADO....358
Creating the Application Shell....358
Building a Custom Record Class....361
Connecting and Retrieving Data....365
Populating the Form....367
Saving Updates....370
Navigating the Record Set....371
Adding New Records....373
Deleting Records....375
Summary....376
Q&A....377
Workshop....377
Quiz....377
Exercise....378

DAY 16 CREATING YOUR OWN CLASSES AND MODULES 379

Designing Classes....380
Encapsulation....380
Inheritance....380
Visual C++ Class Types....381
Creating Library Modules....382
Using Library Modules....383
Creating the Library Module....383
Creating a Test Application....393
Updating the Library Module....398

Summary 401
Q&A 402
Workshop 403
Quiz 403
Exercise 403

DAY 17 SHARING YOUR FUNCTIONALITY WITH OTHER APPLICATIONS—CREATING DLLS 405

Why Create DLLs? 406
Creating and Using DLLs 406
Designing DLLs 409
Creating and Using an MFC Extension DLL 410
Creating the MFC Extension DLL 410
Adapting the Test Application 413
Changing the DLL 414
Creating and Using a Regular DLL 416
Creating the Regular DLL 417
Adapting the Test Application 423
Summary 426
Q&A 427
Workshop 428
Quiz 428
Exercises 428

DAY 18 DOING MULTIPLE TASKS AT ONE TIME—MULTITASKING 429

What Is Multitasking? 430
Performing Multiple Tasks at One Time 430
Idle Processing Threads 431
Spawning Independent Threads 431
Building a Multitasking Application 441
Creating a Framework 441
Designing Spinners 443
Supporting the Spinners 449
Adding the `OnIdle` Tasks 455
Adding Independent Threads 460
Summary 468
Q&A 468
Workshop 471
Quiz 471
Exercises 471

DAY 19 BUILDING YOUR OWN WIDGETS—CREATING ACTIVEX CONTROLS 473

What Is an ActiveX Control? 474
Properties 474
Methods 475
Events 476
Creating an ActiveX Control 476
Building the Control Shell 477
Modifying the `CModArt` Class 478
Adding Properties 481
Designing and Building the Property Page 483
Adding Basic Control Functionality 485
Adding Methods 487
Adding Events 489
Testing the Control 491
Summary 493
Q&A 493
Workshop 494
Quiz 494
Exercises 494

DAY 20 INTERNET APPLICATIONS AND NETWORK COMMUNICATIONS 495

How Do Network Communications Work? 496
Sockets, Ports, and Addresses 497
Creating a Socket 498
Making a Connection 500
Sending and Receiving Messages 501
Closing the Connection 503
Socket Events 503
Detecting Errors 504
Building a Networked Application 505
Creating the Application Shell 505
Window Layout and Startup Functionality 505
Inheriting from the `CAsyncSocket` Class 509
Connecting the Application 511
Sending and Receiving 515
Ending the Connection 517
Summary 519
Q&A 519
Workshop 520
Quiz 520
Exercise 520

Day 21 Adding Web Browsing Functionality to Your Applications 521
The Internet Explorer ActiveX Model 522
The `CHtmlView` Class 523
Navigating the Web 523
Controlling the Browser 524
Getting the Browser Status 524
Building a Web-Browsing Application....525
Creating the Application Shell 525
Adding Navigation Functionality 526
Summary....535
Q&A 535
Workshop 536
Quiz 536
Exercises 536

Week 3 in Review 537

Appendixes 541

Appendix A C++ Review 541
Creating Your First Application 541
`Helloworld.cpp` 543
The `if` Statement, Operators, and Polymorphism 549
Global and Local Variables 552
Pointers 554
References 557
Classes....559
Constructors and Destructors 564
Inheritance....569
Summary....577

Appendix B Answers 579
Day 1....579
Quiz 579
Exercise 580
Day 2....580
Quiz 580
Exercises 581
Day 3....585
Quiz 585
Exercises 585
Day 4....591
Quiz 591
Exercise 591
Day 5....593

Quiz 593
Exercises 594
Day 6....................596
Quiz 596
Exercises 596
Day 7....................597
Quiz 597
Exercises 598
Day 8....................602
Quiz 602
Exercises 602
Day 9....................604
Quiz 604
Exercise 604
Day 10....................605
Quiz 605
Exercise 606
Day 11....................612
Quiz 612
Exercise 612
Day 12....................614
Quiz 614
Exercises 615
Day 13....................616
Quiz 616
Exercise 617
Day 14....................620
Quiz 620
Exercise 621
Day 15....................623
Quiz 623
Exercise 624
Day 16....................625
Quiz 625
Exercises 625
Day 17....................626
Quiz 626
Exercises 626
Day 18....................628
Quiz 628
Exercises 628
Day 19....................631

Quiz 631
Exercises 631
Day 20....632
Quiz 632
Exercises 633
Day 21....634
Quiz 634
Exercises 634

APPENDIX C PRINTING AND PRINT PREVIEWING 637

Using the Framework's Functionality 637
Using Default Print Functionality 638
Overriding `OnPrint()` 641
Using the Printer Device Context 643
Maintaining the Aspect Ratio 645
Pagination and Orientation 647
Setting the Start and End Pages 647
Using the Print Dialog Box 650
Using Portrait and Landscape Orientations....654
Adding GDI Objects with `OnBeginPrinting()` 654
Customizing Device Context Preparation 656
Aborting the Print Job 657
Direct Printing Without the Framework 657
Invoking the Print Dialog Box Directly 657
Using `StartDoc()` and `EndDoc()` 659
Using `StartPage()` and `EndPage()` 660

APPENDIX D UNDERSTANDING AND EXCEPTION HANDLING 661

Using Exceptions 661
Running Code and Catching the Errors 661
Throwing Exceptions....666
Deleting Exceptions....668
MFC Exception Types 668
Using the `CException` Base Class 668
Using the Memory Exception 669
Using the Resource Exceptions....670
Using the File and Archive Exceptions....670
Using the Database Exceptions....672
Using OLE Exceptions 673
Using the Not Supported Exception 674
Using the User Exception 674
Generating Your Own Custom Exception Classes 675

APPENDIX E USING THE DEBUGGER AND PROFILER 677

Creating Debugging and Browse Information 677
Using Debug and Release Modes 678
Setting Debug Options and Levels 679
Creating and Using Browse Information 681
Using Remote and Just-in-Time Debugging 683
Tracing and Single Stepping 685
Using the `TRACE` Macro 685
Using the `ASSERT` and `VERIFY` macros 687
Using Breakpoints and Single Stepping the Program 689
Using Edit and Continue 692
Watching Program Variables 692
Other Debugger Windows 694
Additional Debugging Tools 694
Using Spy++ 695
Process Viewer 697
The OLE/COM Object Viewer 697
The MFC Tracer 698

APPENDIX F USING MFC'S HELPER CLASSES 699

Using the Collection Classes 699
Using the Array Classes 700
Using the List Classes 702
Using the Map Classes 704
Creating Custom Collection Classes 707
Using the Coordinate-Handling Classes 710
Using the `CPoint` Class 710
Using the `CRect` Class 712
Using the `CSize` Class 717
Using the Time-Handling Classes 718
Using the `COleDateTime` Class 719
Using the `COleDateTimeSpan` Class 722
Using the String Manipulation Class 724
Using the `CString` Class 724
String Manipulation 727
Searching Strings 727
Formatting Text for Display 728

INDEX 731

About the Authors

Davis Chapman first began programming computers while working on his master's degree in music composition. Writing applications for computer music, he discovered that he enjoyed designing and developing computer software. It wasn't long before he came to the realization that he stood a much better chance of eating if he stuck with his new-found skill and demoted his hard-earned status as a "starving artist" to a part-time hobby. Since that time, Davis has focused on the art of software design and development, with a strong emphasis on the practical application of client/server technology. Davis is the lead author of *Web Development with Visual Basic 5* and *Building Internet Applications with Delphi 2*. Davis is also a contributing author of *Special Edition Using Active Server Pages* and *Running a Perfect Web Site, Second Edition*. He has been a consultant working and living in Dallas, Texas, for the past eight years, and he can be reached at `davischa@onramp.net`.

Contributing Author

Jon Bates has worked on a whole range of commercial, industrial, and military software development projects worldwide over the past fifteen years. He is currently working as a self-employed software design consultant and contract software developer, specializing in Visual C++ application development for Windows NT/95/98.

Jon began his career writing computer games for popular microcomputers and has since worked with a number of operating systems, such as CPM, DOS, TRIPOS, UNIX, and Windows, and a number of Assembly, third-generation, and object-oriented languages.

He has written system and application software as diverse as device drivers, email, production modeling, motion video, image analysis, network and telecommunications, data capture, control systems, estimating and costing, and visualization software. He has also written a number of technical articles for computing journals on a range of topics.

Jon lives with his wife, Ruth, and dog, Chaos, in the middle of cool Britannia. When not playing with computers, he likes to sleep and dream of fractals.

You can reach Jon at `jon@chaos1.demon.co.uk` and visit his Web site at `www.chaos1.demon.co.uk`.

Dedication

To Dore, and the rest of my family, for being very patient with me while I was busy writing yet another book.

Acknowledgments

There are numerous people without whom this book might never have been written. Among those who deserve credit is Kelly Marshall, for enabling me to take on this project and for sticking with me even though I know I made her life stressful at times. Credit needs to go to the entire editing team at Macmillan. I've seen what some of the material you have to work with looks like when it comes in from the authors, and I don't want to trade jobs with any of you. I'd also like to thank my family for continuing to allow me to put in the work required to produce this book—and for not disowning me in the process.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As the executive editor for the Programming team at Macmillan Computer Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-817-7070

Email: `adv_prog@mcp.com`

Mail: Executive Editor
Programming
Macmillan Computer Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

Welcome to Visual C++. Over the next 21 days, you will learn how to use the features that Microsoft has built into its C++ development environment to enable you to create very advanced applications for the Windows and NT platforms. When Microsoft's developers first came up with the idea behind Visual C++, they decided to take their world-class C++ compiler and create a development environment and set of tools that would enable developers to create Windows applications with a level of ease and speed that was unheard of among C++ development environments. Since that first version, Microsoft has continued to improve the tools that are a part of Visual C++ to make it even easier to create Windows applications. As Microsoft has introduced new technologies into the Windows platforms, it has also introduced tools into the Visual C++ suite to make it easy to integrate these new technologies into your applications.

If you are new to C++, don't worry. I've tried to make it easy for you to learn the C++ programming language while also learning how to build applications using the Visual C++ tools. Just in case you find yourself having trouble understanding some aspect of C++, I've included a brief overview of the programming language in Appendix A, "C++ Review."

If you've looked at previous versions of this book, you might notice that I've completely rewritten the entire book. Our goal with this new version is not just to introduce you to and guide you through the various tools and resources that you will use to build applications with Visual C++; I've also tried to include a great deal more detail about the various options that are available to you with each of the features that are covered. This way, you'll be able to get a lot of use out of this book long after the initial 21 days.

How This Book Is Organized

This book is organized in weeks, with each set of seven days set off into a part unto itself. However, even though the book is organized in weeks, the topics are not necessarily organized that way.

For the first week, you'll be covering the basics of building applications with Visual C++. You'll learn how to use designers to design your application windows. You'll learn how to use various controls available to you as a Windows application developer. You'll also learn a lot about the Visual C++ development environment and the tools that it makes available to you.

By the time you begin the second week, you'll be doing more and more programming, as the topics become more involved. You'll still be using the Visual C++ tools to construct your applications, but the programming code will be getting a little more involved. You'll also start learning about more advanced topics, such as displaying graphics and creating SDI and MDI applications. Toward the end of the second week, you'll begin to work with databases. This topic spills over into the third and final week.

In the third week, you'll learn how to create your own modules, DLLs, and ActiveX controls. You'll also learn how to build multitasking applications, which perform multiple tasks at a time. Finally, you'll learn how to integrate Microsoft Internet Explorer, and the ActiveX controls it provides, into your applications so that you can extend your applications over the Internet.

After you finish the third week, you'll be ready to tackle the world of Windows programming with Visual C++. You'll have the skills and know-how required to build most Windows applications available today.

Conventions Used in This Book

While you are reading this book, you will probably notice a couple conventions that have been used to make it easier for you to learn the topic being discussed.

All the source code in this book is provided in a `monospaced` font, as shown in Listing 0.1. This includes all the source code from the applications that you will be building and illustrations of how various functions can be used. Whenever you are adding new code, or changing code in a function with other code already there, the line numbers of the code that you add or change will be pointed out in the text.

LISTING 0.1. SOME SAMPLE CODE.

```
1: void main()
2: {
3:     // if you are adding or changing code in an existing
4:     // code snippet, I will point out the line numbers in the text.
5: }
```

If a topic needs special attention, it will be set apart from the rest of the text by one of several special markers:

- Notes
- Tips
- Cautions

Notes offer a deeper explanation of a topic or explain interesting or important points.

Tips are pieces of information that can make things easier.

Cautions warn you about traps that you will want to avoid.

At the end of each day, you'll find a short quiz and one or two exercises to help make sure that you learned the topic you were studying. Don't worry—just in case you need the answers to the quizzes and some guidance when building the exercises, the solutions are provided in Appendix B, "Answers."

Enough said! You didn't buy this book to read about this book. You bought this book to learn how to use Visual C++ to build Windows applications. So go ahead and flip the page and get started programming…

WEEK 1

At a Glance

Welcome to the world of Visual C++. Over the next three weeks, you'll learn how to build a wide variety of applications using this extremely flexible and complete programming tool. Each day you'll learn about a different area of functionality and how you can use it in your applications. What's more—every one of the areas of functionality will be accompanied with a hands-on sample application that you will build yourself. There's not a more effective way of learning new technologies than to work with them yourself. Learning by doing…that's what you'll do as you make your way through this book.

Over the course of the first week, you'll learn about several of the basics that are involved in building applications with Visual C++. This starts on the first day as you learn about and become familiar with the Visual C++ development environment by building a simple application.

On Day 2, you'll begin learning more about the specifics of building applications in Visual C++. You'll learn about the standard controls that are used in Windows applications, how you can place and configure these on an application window, and how you can interact with them.

On Day 3, you'll learn how you can capture mouse and keyboard events and react to them in your applications. You'll see how you can determine where the mouse is in your application space. You'll also learn how to determine what keys the user is pressing on the keyboard and how you can react to these user actions.

On Day 4, you'll learn how to work with timers in a Visual C++ application. You'll learn how to have two or more timers running at the same time and how you can tell them apart.

On Day 5, you'll see how you can add additional windows to your application and how you can use them to get information from the user. You'll see how you can use built-in dialogs to ask the user simple questions and how you can build your own custom dialogs to get more detailed information.

On Day 6, you'll learn how to create menus to add to your application. You'll see how you can call functions in your application from menus that you have added to your application.

On Day 7, you'll learn about the font infrastructure in Windows and how you can access it in your Visual C++ applications. You'll see how you can build a list of available fonts and how you can display text in any of these fonts.

That will end the first week of this book. At that time, you can look back over what you have learned during the week and think about all that you can do with what you have learned when you build applications. So, without further ado, go ahead and jump in and get started.

WEEK 1

DAY 1

The Visual C++ Development Environment—Building Your First Visual C++ Application

Welcome to *Sams Teach Yourself Visual C++ 6 in 21 Days*. Over the next three weeks, you will learn how to build a wide variety of applications with Microsoft's Visual C++. What's even better is that you will learn how to create these types of applications by actually building them yourself. As you read this book, you will be gaining actual programming experience using Visual C++. So let's get started!

Today, your focus will be on learning about the Visual C++ development environment and some of the tools that it provides for building applications. Although Visual C++ provides more tools than you would probably use in any

one application development effort—even more than you could possibly learn to use in a single day—I limit the focus to the primary tools that you will use throughout this book, as well as in just about every application you build with Visual C++. Today, you'll learn about the following:

- The primary areas of the Visual C++ development environment
- The Application Wizard—how you can use it to build the basic infrastructure for your applications
- The Dialog Painter—how you can use it to paint dialog windows, much in the same way that you can build windows with Visual Basic, PowerBuilder, or Delphi
- The Class Wizard—how you can use it to attach functionality to your application windows

The Visual C++ Development Environment

Before you begin your quick tour around the Visual C++ development environment, you should start Visual C++ on your computer so that you can see firsthand how each of the areas are arranged and how you can change and alter that arrangement yourself.

After Developer Studio (the Microsoft Visual development environment) starts, you see a window that looks like Figure 1.1. Each of the areas has a specific purpose in the Developer Studio environment. You can rearrange these areas to customize the Developer Studio environment so that it suits your particular development needs.

The Workspace

When you start Visual C++ for the first time, an area on the left side of Developer Studio looks like it is taking up a lot of real estate and providing little to show for it. This area is known as the workspace, and it is your key to navigating the various pieces and parts of your development projects. The workspace allows you to view the parts of your application in three different ways:

- Class View allows you to navigate and manipulate your source code on a C++ class level.
- Resource View allows you to find and edit each of the various resources in your application, including dialog window designs, icons, and menus.
- File View allows you to view and navigate all the files that make up your application.

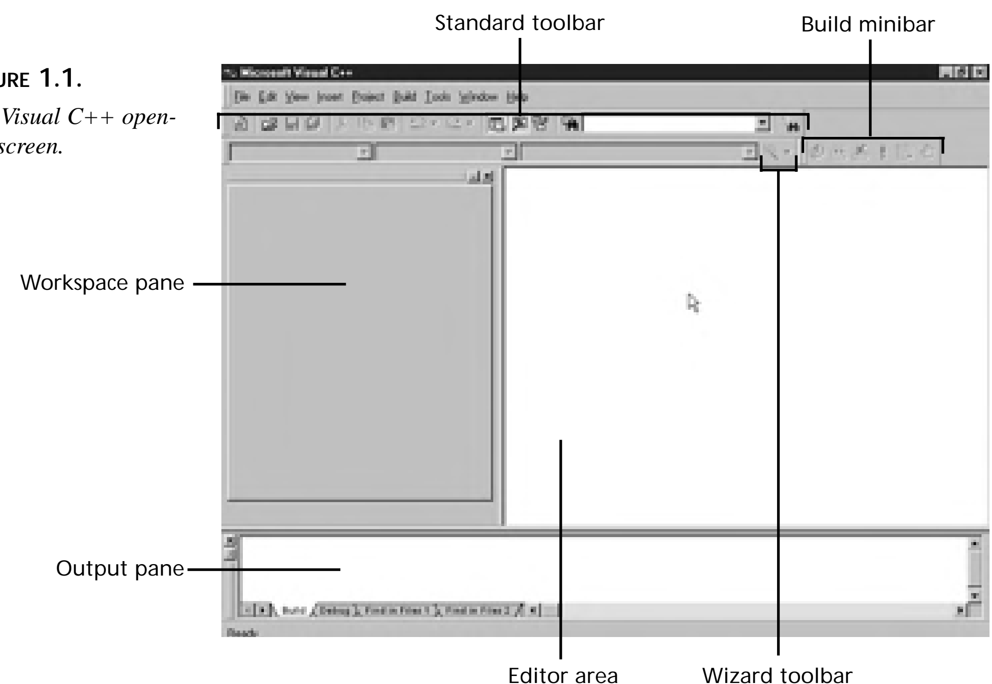


FIGURE 1.1.
The Visual C++ opening screen.

The Output Pane

The Output pane might not be visible when you start Visual C++ for the first time. After you compile your first application, it appears at the bottom of the Developer Studio environment and remains open until you choose to close it. The Output pane is where Developer Studio provides any information that it needs to give you; where you see all the compiler progress statements, warnings, and error messages; and where the Visual C++ debugger displays all the variables with their current values as you step through your code. After you close the Output pane, it reopens itself when Visual C++ has any message that it needs to display for you.

The Editor Area

The area on the right side of the Developer Studio environment is the editor area. This is the area where you perform all your editing when using Visual C++, where the code editor windows display when you edit C++ source code, and where the window painter displays when you design a dialog box. The editor area is even where the icon painter displays when you design the icons for use in your applications. The editor area is basically the entire Developer Studio area that is not otherwise occupied by panes, menus, or toolbars.

Menu Bars

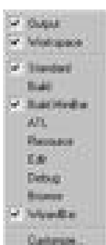
The first time you run Visual C++, three toolbars display just below the menu bar. Many other toolbars are available in Visual C++, and you can customize and create your own toolbars to accommodate how you best work. The three toolbars that are initially open are the following:

- The Standard toolbar contains most of the standard tools for opening and saving files, cutting, copying, pasting, and a variety of other commands that you are likely to find useful.
- The WizardBar toolbar enables you to perform a number of Class Wizard actions without opening the Class Wizard.
- The Build minibar provides you with the build and run commands that you are most likely to use as you develop and test your applications. The full Build toolbar also lets you switch between multiple build configurations (such as between the Debug and Release build configurations).

Rearranging the Developer Studio Environment

The Developer Studio provides two easy ways to rearrange your development environment. The first is by right-clicking your mouse over the toolbar area. This action opens the pop-up menu shown in Figure 1.2, allowing you to turn on and off various toolbars and panes.

FIGURE 1.2.
Toolbar on and off menu.



Another way that you can easily rearrange your development environment is to grab the double bars at the left end of any of the toolbars or panes with the mouse. You can drag the toolbars away from where they are currently docked, making them floating toolbars, as in Figure 1.3. You can drag these toolbars (and panes) to any other edge of the Developer Studio to dock them in a new spot. Even when the toolbars are docked, you can use the double bars to drag the toolbar left and right to place the toolbar where you want it to be located.

FIGURE 1.3.
Example of a floating minibar.

> On the workspace and Output panes, the double bars that you can use to drag the pane around the Developer Studio environment might appear on the top of the pane or on the left side, depending on how and where the pane is docked.

Starting Your First Project

For your first Visual C++ application, you are going to create a simple application that presents the user with two buttons, as in Figure 1.4. The first button will present the user with a simple greeting message, shown in Figure 1.5, and the second button will close the application. In building this application, you will need to do the following things:

1. Create a new project workspace.
2. Use the Application Wizard to create the application framework.
3. Rearrange the dialog that is automatically created by the Application Wizard to resemble how you want the application to look.
4. Add the C++ code to show the greeting to the user.
5. Create a new icon for the application.

FIGURE 1.4.
Your first Visual C++ application.

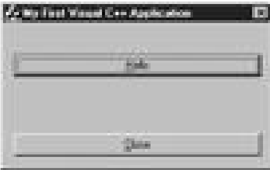


FIGURE 1.5.
If the user clicks the first button, a simple greeting is shown.

Creating the Project Workspace

Every application development project needs its own project workspace in Visual C++. The workspace includes the directories where the application source code is kept, as well

as the directories where the various build configuration files are located. You can create a new project workspace by following these steps:

1. Select File | New. This opens the New Wizard shown in Figure 1.6.

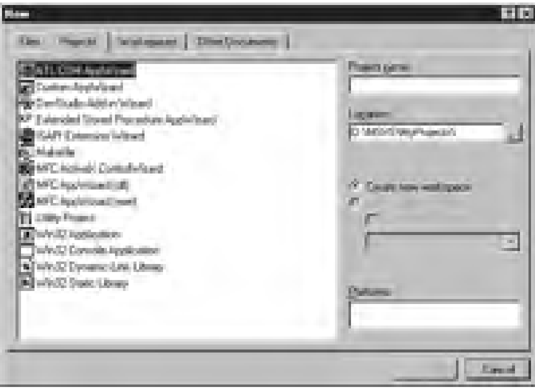


FIGURE 1.6.
The New Wizard.

2. On the Projects tab, select MFC AppWizard (exe).
3. Type a name for your project, such as Hello, in the Project Name field.
4. Click OK. This causes the New Wizard to do two things: create a project directory (specified in the Location field) and then start the AppWizard.

Using the Application Wizard to Create the Application Shell

The AppWizard asks you a series of questions about what type of application you are building and what features and functionality you need. It uses this information to create a shell of an application that you can immediately compile and run. This shell provides you with the basic infrastructure that you need to build your application around. You will see how this works as you follow these steps:

1. In Step 1 of the AppWizard, specify that you want to create a Dialog-based application. Click Next at the bottom of the wizard.
2. In Step 2 of the AppWizard, the wizard asks you about a number of features that you can include in your application. You can uncheck the option for including support for ActiveX controls if you will not be using any ActiveX controls in your application. Because you won't be using any ActiveX controls in today's application, go ahead and uncheck this box.
3. In the field near the bottom of the wizard, delete the project name (Hello) and type in the title that you want to appear in the title bar of the main application window,

such as **`My First Visual C++ Application`**. Click Next at the bottom of the wizard.

4. In Step 3 of the AppWizard, leave the defaults for including source file comments and using the MFC library as a DLL. Click Next at the bottom of the wizard to proceed to the final AppWizard step.
5. The final step of the AppWizard shows you the C++ classes that the AppWizard will create for your application. Click Finish to let AppWizard generate your application shell.
6. Before AppWizard creates your application shell, it presents you with a list of what it is going to put into the application shell, as shown in Figure 1.7, based on the options you selected when going through the AppWizard. Click OK and AppWizard generates your application.

FIGURE 1.7.

The New Project Information screen.

7. After the AppWizard generates your application shell, you are returned to the Developer Studio environment. You will notice that the workspace pane now presents you with a tree view of the classes in your application shell, as in Figure 1.8. You might also be presented with the main dialog window in the editor area of the Developer Studio area.
8. Select Build | Build Hello.exe to compile your application.
9. As the Visual C++ compiler builds your application, you see progress and other compiler messages scroll by in the Output pane. After your application is built, the Output pane should display a message telling you that there were no errors or warnings, as in Figure 1.9.

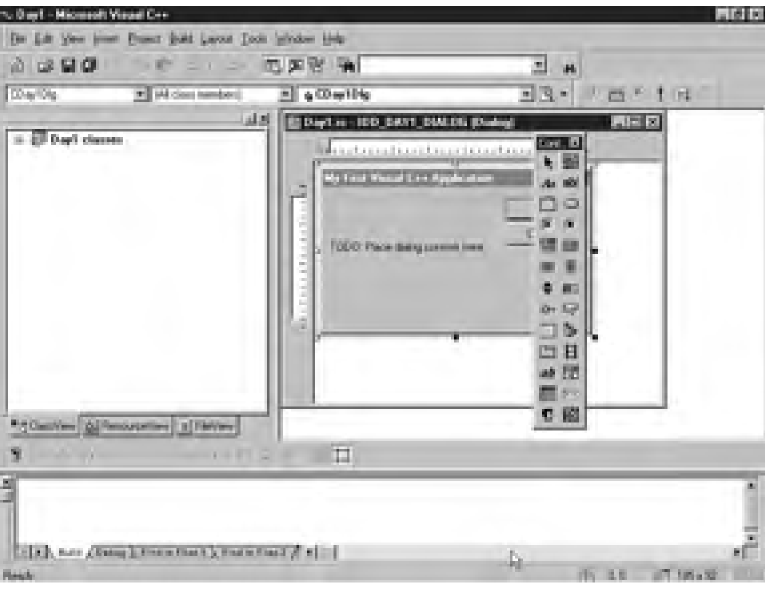


FIGURE 1.8.
Your workspace with a tree view of the project's classes.

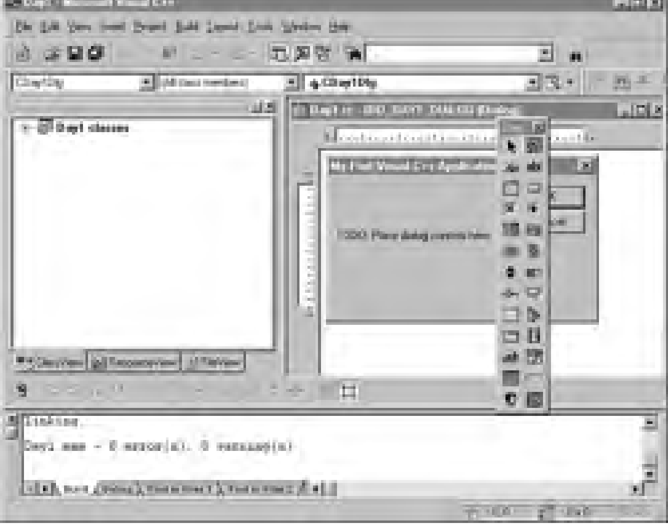


FIGURE 1.9.
The Output pane displays any compiler errors.

10. Select Build | Execute Hello.exe to run your application.
11. Your application presents a dialog with a `TODO` message and OK and Cancel buttons, as shown in Figure 1.10. You can click either button to close the application.

FIGURE 1.10.
The unmodified application shell.

Designing Your Application Window

Now that you have a running application shell, you need to turn your focus to the window layout of your application. Even though the main dialog window may already be available for painting in the editor area, you should still navigate to find the dialog window in the workspace so that you can easily find the window in subsequent development efforts. To redesign the layout of your application dialog, follow these steps:

1. Select the Resource View tab in the workspace pane, as in Figure 1.11.

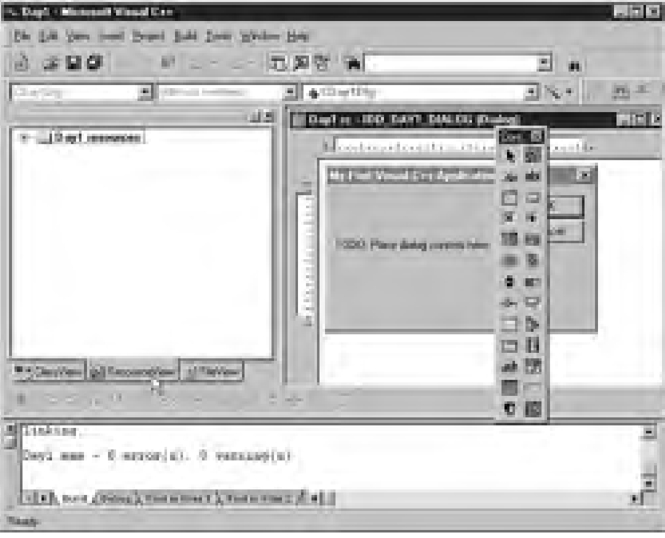


FIGURE 1.11.
The Resource View tab in the workspace pane.

2. Expand the resources tree to display the available dialogs. At this point, you can double-click the `IDD_DAY1_DIALOG` dialog to open the window in the Developer Studio editor area.
3. Select the text displayed in the dialog and delete it using the Delete key.
4. Select the Cancel button, drag it down to the bottom of the dialog, and resize it so that it is the full width of the layout area of the window, as in Figure 1.12.

FIGURE 1.12.
Positioning the Cancel button.

5. Right-click the mouse over the Cancel button, opening the pop-up menu in Figure 1.13. Select Properties from the menu, and the properties dialog in Figure 1.14 opens.

FIGURE 1.13.
Right-clicking the mouse to open a pop-up menu.

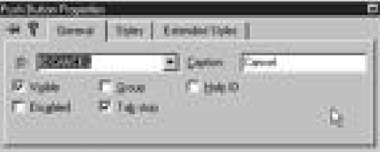


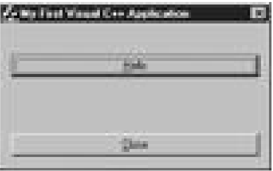
FIGURE 1.14.
The Cancel button properties dialog.

6. Change the value in the Caption field to `&Close`. Close the properties dialog by clicking the Close icon in the upper-right corner of the dialog.
7. Move and resize the OK button to around the middle of the window, as in Figure 1.15.

FIGURE 1.15.
Positioning the OK button.

8. On the OK button properties dialog, change the ID value to `IDHELLO` and the caption to `&Hello`.
9. Now when you compile and run your application, it will look like what you've just designed, as shown in Figure 1.16.

FIGURE 1.16.
Running your redesigned application.



If you play with your application, you will notice that the Close button still closes the application. However, the Hello button no longer does anything because you changed the ID of the button. MFC applications contain a series of macros in the source code that determine which functions to call based on the ID and event message of each control in the application. Because you changed the ID of the Hello button, these macros no longer know which function to call when the button is clicked.

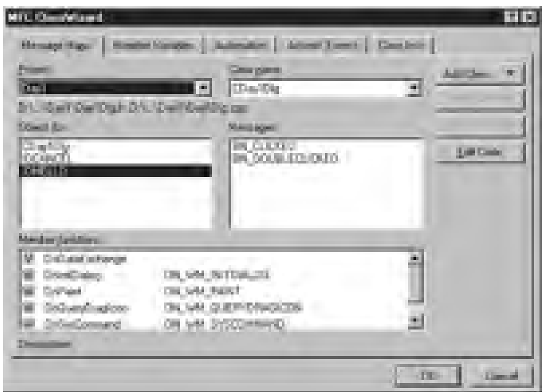
Adding Code to Your Application

You can attach code to your dialog through the Visual C++ Class Wizard. You can use the Class Wizard to build the table of Windows messages that the application might receive, including the functions they should be passed to for processing, that the MFC macros use for attaching functionality to window controls. You can attach the functionality for this first application by following these steps:

1. To attach some functionality to the Hello button, right-click over the button and select Class Wizard from the pop-up menu.
2. If you had the Hello button selected when you opened the Class Wizard, it is already selected in the list of available Object IDs, as in Figure 1.17.

FIGURE 1.17.

The Class Wizard.



3. With `IDHELLO` selected in the Object ID list, select `BN_CLICKED` in the list of messages and click Add Function. This opens the Add Member Function dialog shown in Figure 1.18. This dialog contains a suggestion for the function name. Click OK to create the function and add it to the message map.

FIGURE 1.18.

The Class Wizard Add Member Function dialog.

4. After the function is added for the click message on the Hello button, select the `OnHello` function in the list of available functions, as in Figure 1.19. Click the Edit Code button so that your cursor is positioned in the source code for the function, right at the position where you should add your functionality.

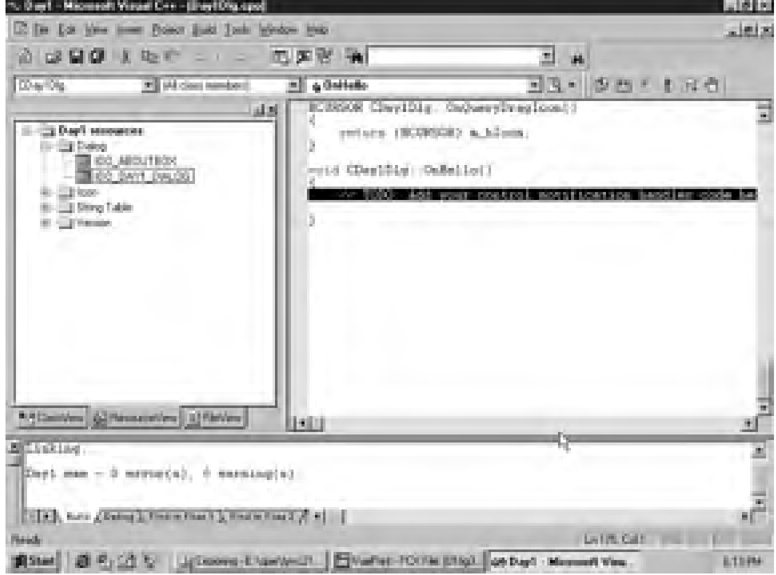
FIGURE 1.19.

The list of available functions in the Class Wizard.

5. Add the code in Listing 1.1 just below the `TODO` comment line, as shown in Figure 1.20.

FIGURE 1.20.

Source code view where you insert Listing 1.1.



LISTING 1.1. `HELLODLG.CPP`—THE `OnHello` FUNCTION.

```
 1: Void CHelloDlg::OnHello()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Say hello to the user
10:     MessageBox("Hello. This is my first Visual C++ Application!");
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
```

6. When you compile and run your application, the Hello button should display the message shown in Figure 1.21.

FIGURE 1.21.
Now your application will say hello to you.

Finishing Touches

Now that your application is functionally complete, you can still add a few details to finish off the project. Those finishing touches include

- Creating the dialog box icon
- Adding maximize and minimize buttons

Creating the Dialog Box Icon

If you noticed the icon in the top-left corner of your application window, you saw three blocks with the letters M, F, and C. What does MFC have to do with your application? MFC stands for Microsoft Foundation Classes. Technically, it's the C++ class library that your application is built with, but do you want to broadcast that to every user who sees your application? Most likely not. You need to edit the application icon to display an image that you do want to represent your application. Let's get busy!

1. In the tree view of your application resources in the workspace pane, expand the icon branch and select the `IDR_MAINFRAME` icon, as in Figure 1.22. This brings the application icon into the editor area of the Developer Studio.

FIGURE 1.22.
The standard MFC icon.

2. Using the painting tools provided, repaint the icon to display an image that you want to use to represent your application, as in Figure 1.23.

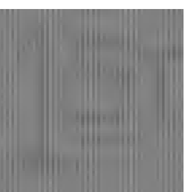


FIGURE 1.23.
Your own custom icon for your application.

3. When you compile and run your application, you will notice your custom icon in the top-left corner of your application window. Click the icon and select About Hello from the drop-down menu.
4. On the About dialog that Visual C++ created for you, you can see a large version of your custom icon in all its glory, as shown in Figure 1.24.

FIGURE 1.24.
Your application's About window.

When you open an application icon in the icon designer, the icon is sized by default at 32×32. You can also select a 16×16 size icon from the drop-down list box just above where you are drawing the icon. You should draw both of these icons because there are some instances in which the large icon will be displayed and some instance in which the small icon will be shown. You will want both icons to show the same image to represent your application.

Adding Maximize and Minimize Buttons

In the dialog editor, where you design your application window, you can add the minimize and maximize buttons to the title bar of your application window by following these steps:

1. Select the dialog window itself as if you were going to resize the window.
2. Using the pop-up menu (from right-clicking the mouse), select the dialog properties.
3. Select the Styles tab, as shown in Figure 1.25.

FIGURE 1.25.
Turning the minimize and maximize buttons on and off.

4. After you turn on the minimize and maximize boxes, you can compile and run your application. The minimize and maximize buttons appear on the title bar, as in Figure 1.26.

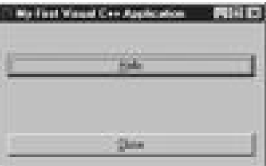


FIGURE 1.26.
The application window with the minimize and maximize buttons.

Summary

Today you got your first taste of building applications using Visual C++. You learned about the different areas of the Visual C++ Developer Studio and what function each of these areas serves. You also learned how you can rearrange the Developer Studio environment to suit the way you work. You also learned how you can use the Visual C++ wizards to create an application shell and then attach functionality to the visual components that you place on your application windows.

Q&A

Q How can I change the title on the message box, instead of using the application name?

A By default, the message box window uses the application name as the window title. You can change this by adding a second text string to the `MessageBox` function call. The first string is always the message to be displayed, and the second string is used as the window title. For example, the `OnHello` function would look like

```
// Say hello to the user
MessageBox("Hello. This is my first Visual C++ Application!",
           "My First Application");
```

Q Can I change the text on the About window to give my company name and more detailed copyright information?

A Yes, the About window is in the Dialogs folder in the Resources View tab of the workspace pane. If you double-click the `IDD_ABOUTBOX` dialog, the About box will be opened in the dialog designer, where you can redesign it however you want.

Workshop

The Workshop provides quiz questions to help solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. How do you change the caption on a button?
2. What can you do with the Visual C++ AppWizard?
3. How do you attach functionality to the click of a button?

Exercise

Add a second button to the About window in your application. Have the button display a different message from the one on the first window.

WEEK 1

DAY 2

Using Controls in Your Application

Some of the things that you will find in just about every Windows application are buttons, check boxes, text fields, and drop-down list boxes. These are known as controls, and many of these controls are built into the operating system itself. With Visual C++, using these common controls is as easy as placing them on a dialog window with a drag-and-drop window design method. Today you are going to learn

- What the basic controls in Visual C++ are
- How to declare and attach variables to a controls
- How to synchronize the values between a control and a variable
- How to specify the order users navigate around your application windows
- How to trigger actions with controls
- How to manipulate and alter the appearance of controls (while your application is running)

The Basic Windows Controls

Several standard controls are built into the Windows operating system, including such things as sliders, tree and list controls, progress bars, and so on. However, today you will work with a half dozen controls that appear in just about every Windows application:

- Static text
- Edit box
- Command button
- Check box
- Radio button
- Drop-down list box (also known as a combo box)

These and other controls are readily available for use in Visual C++ applications. They can be found on the controls palette in the Dialog Painter editor in the Developer Studio, as shown in Figure 2.1.

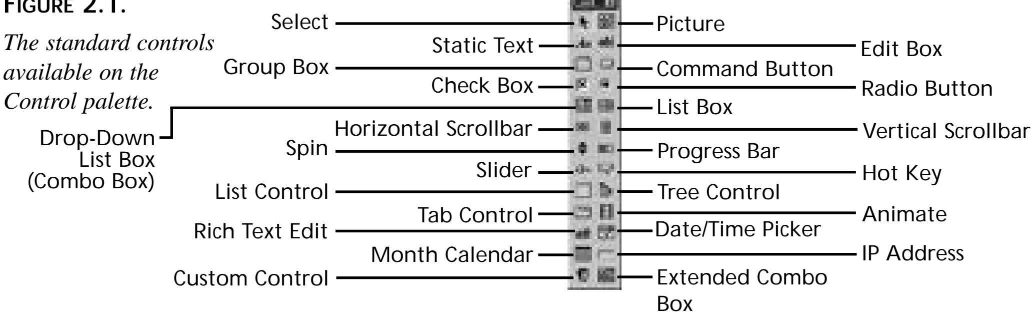


FIGURE 2.1.
The standard controls available on the Control palette.

The Static Text Control

You use the static text control to present text to the user. The user will not be able to change the text or otherwise interact with the control. Static text is intended as a read-only control. However, you can easily change the text displayed by the control as your application is running through the code you create for your application.

The Edit Box Control

An edit box allows the user to enter or change text. The edit box is one of the primary tools for allowing the user to enter specific information that your application needs. It is a control that allows the user to type a specific amount of text, which you can capture

and use for any needed purpose. The edit box accepts plain text only; no formatting is available to the user.

The Command Button Control

A command button is a button that the user can press to trigger some action. Command buttons have a textual label that can give users some idea of what will happen when they click that button. Buttons can also have images as part of the button, allowing you to place an image on the button—alone or along with a textual description—to convey what the button does.

The Check Box Control

A check box is a square that the user can click to check (×) or uncheck. The check box control is used to turn a particular value on and off. They are basically on/off switches with an occasional third, in-between state. You normally use check boxes to control discrete, on/off-type variables.

The Radio Button Control

A radio button is a circle that the user can click to fill with a black spot. The radio button is similar to the check box control, but it is used in a group of two or more where only one of the values can be in the on state at a time. You normally use radio buttons in groups of at least three, surrounded by a group box. The group box allows each group of radio buttons to be independent so that only one radio button in each group can be in the on state at any time.

The Drop-Down List Box Control

A drop-down list box, or combo control, is an edit box with a list of available values attached. You use the drop-down list box to provide a list of choices, from which the user may select one value from the list. Sometimes, the user is given the option of typing in his own value when a suitable one isn't provided in the list.

Adding Controls to Your Window

The application you are going to build today will have a number of controls on a single dialog window, as shown in Figure 2.2. These controls have a number of different functions. At the top of the window is an edit field where the user can enter a message that displays in a message box when he or she clicks the button beside the field. Below this edit field are two buttons that either populate the edit field with a default message or clear the edit field. Below these buttons is a drop-down list box that contains a list of

standard Windows applications. When the user selects one of these programs and then clicks the button beside the drop-down list, the selected program will run. Next are two groups of check boxes that affect the controls you add to the top half of the dialog: the controls for displaying a user message and the controls for running another program. The left set of check boxes will enable and disable each group of controls you provide. The right set of check boxes will show and hide each group of controls. At the bottom of the dialog box is a button that can be clicked to close the application.

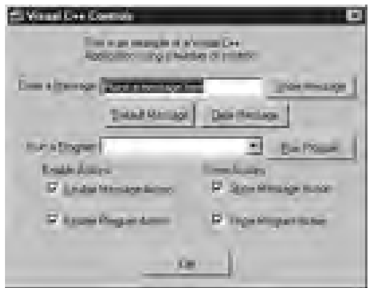


FIGURE 2.2.
Today's application will use a number of standard controls.

Creating the Application Shell and Dialog Layout

Using what you learned yesterday, create a new application shell and design the application dialog layout as follows:

1. Create a new AppWizard workspace project, calling the project **`Day2`**.
2. Use the same settings in the AppWizard as you used yesterday; specify the dialog title **`Visual C++ Controls`**.
3. After you create the application shell, lay out the main dialog as shown earlier in Figure 2.2.
4. Configure the control properties as specified in Table 2.1.

TABLE 2.1. PROPERTY SETTINGS FOR THE CONTROLS ON THE APPLICATION DIALOG.

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`This is an example of a Visual C++ Application using a number of controls.`
Static Text	ID	`IDC_STATICMSG`
	Caption	`Enter a &Message:`
Static Text	ID	`IDC_STATICPGM`
	Caption	`Run a &Program:`

Object	*Property*	*Setting*
Edit Box	ID	`IDC_MSG`
Button	ID	`IDC_SHWMSG`
	Caption	`&Show Message`
Button	ID	`IDC_DFLTMSG`
	Caption	`&Default Message`
Button	ID	`IDC_CLRMSG`
	Caption	`&Clear Message`
Button	ID	`IDC_RUNPGM`
	Caption	`&Run Program`
Button	ID	`IDC_EXIT`
	Caption	`E&xit`
Combo Box	ID	`IDC_PROGTORUN`
Group Box	ID	`IDC_STATIC`
	Caption	`Enable Actions`
Group Box	ID	`IDC_STATIC`
	Caption	`Show Actions`
Check Box	ID	`IDC_CKENBLMSG`
	Caption	`&Enable Message Action`
Check Box	ID	`IDC_CKENBLPGM`
	Caption	`E&nable Program Action`
Check Box	ID	`IDC_CKSHWMSG`
	Caption	`S&how Message Action`
Check Box	ID	`IDC_CKSHWPGM`
	Caption	`Sh&ow Program Action`

Tip

When adding a combo box control to the window, it is important that you click and drag the area for the control as large as you want the drop-down list to be. After you draw the control on the window, you can resize the width of the control as you would normally expect to do. To resize how far the list drops down, you need to click the arrow, as if you were trying to trigger the drop-down list while the application was running.

5. After you place all these controls on the dialog window and configure all their properties, reopen the properties dialog for the combo box that you placed on the window. On the Data tab of the properties dialog, enter the following values, using a Control+Enter key combination to add the second and third items, as shown in Figure 2.3.
   - Notepad
   - Paint
   - Solitaire

FIGURE 2.3.
Use the properties dialog to add entries in the combo box's drop-down list.

Specifying the Control Tab Order

Now that you have all the controls laid out on the window, you need to make sure that the user navigates in the order you want if he or she uses the Tab key to move around the window. You can specify the tab order by following these steps:

1. Select either the dialog window or one of the controls on the window in the editing area of the Developer Studio.
2. Choose Layout | Tab Order from the menu. By turning on the Tab Order, you see a number beside each of the controls on the window. The numbers indicate the order in which the dialog will be navigated, as shown in Figure 2.4.

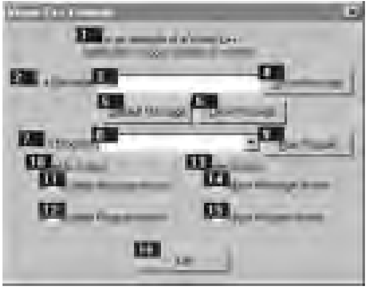


FIGURE 2.4.
Turning on Tab Order shows the order in which the dialog will be navigated.

3. Using the mouse, click each of the number boxes in the order that you want the user to navigate the window. The controls will renumber themselves to match the order in which you selected them.

4. Once you specify the tab order, select Layout | Tab Order once again to return to the layout editor.

Any static text that has a mnemonic should appear just before the control that accompanies the text in the tab order. Because the user cannot interact with the static text, when the user chooses the mnemonic, the focus will go directly to the next control in the tab order.

A mnemonic is the underlined character in the caption on a button, check box, menu, or other control label. The user can press this underlined character and the Alt key at the same time to go directly to that control or to trigger the clicked event on the control. You specify a mnemonic by placing an ampersand (`&`) in front of the character to be used as the mnemonic when you type the Caption value. It is important to make certain that you do not use the same mnemonic more than once on the same window, or set of menus, because the user can get confused when choosing a mnemonic doesn't result in the action that he or she expects.

One last thing that you want to do before getting into the details of the application code is check your mnemonics to make certain that there are no conflicts in your controls. Follow these steps:

1. Select the dialog window or one of the controls in the layout editor. Right-click the mouse and select Check Mnemonics.
2. If there are no conflicts in your mnemonics, Visual C++ returns a message box dialog, letting you know that there are no conflicts (see Figure 2.5).

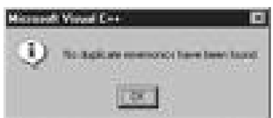


FIGURE 2.5.
The mnemonic checker tells you whether there are conflicts.

3. If any conflicts exist, the dialog indicates the conflicting letter and gives you the option of automatically selecting the controls containing the conflicting mnemonics, as in Figure 2.6.

FIGURE 2.6.
Duplicate mnemonics can be automatically selected.

Attaching Variables to Your Controls

At this point, if you've programmed using Visual Basic or PowerBuilder, you probably figure that you're ready to start slinging some code. Well, with Visual C++, it's not quite the same process. Before you can begin coding, you have to assign variables to each of the controls that will have a value attached—everything except the static text and the command buttons. You will interact with these variables when you write the code for your application. The values that the user enters into the screen controls are placed into these variables for use in the application code. Likewise, any values that your application code places into these variables are updated in the controls on the window for the user to see.

How do you declare these variables and associate them with the controls that you placed on the window? Follow these steps:

1. Open the Class Wizard, as you learned yesterday.
2. Select the Member Variables tab, as shown in Figure 2.7.

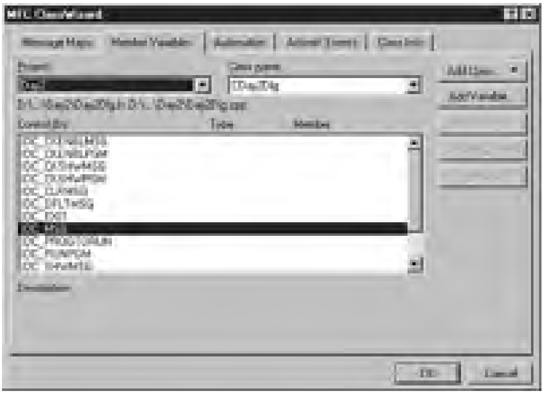


FIGURE 2.7.
The Member Variables tab on the Class Wizard is where you add variables to controls.

3. Select the ID of one of the controls that you need to attach a variable to, such as `IDC_MSG`.
4. Click the Add Variable button.
5. In the Add Member Variable dialog, enter the variable name, specifying the category and variable type, as shown in Figure 2.8. Click OK.
6. Repeat steps 3 through 5 for all the other controls for which you need to add variables. You should add the variables for your application as listed in Table 2.2.

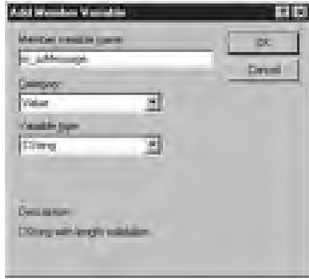


FIGURE 2.8.
Adding a variable to a control.

TABLE 2.2. VARIABLES FOR APPLICATION CONTROLS.

Control	*Variable Name*	*Category*	*Type*
`IDC_MSG`	`m_strMessage`	Value	`CString`
`IDC_PROGTORUN`	`m_strProgToRun`	Value	`CString`
`IDC_CKENBLMSG`	`m_bEnableMsg`	Value	`BOOL`
`IDC_CKENBLPGM`	`m_bEnablePgm`	Value	`BOOL`
`IDC_CKSHWMSG`	`m_bShowMsg`	Value	`BOOL`
`IDC_CKSHWPGM`	`m_bShowPgm`	Value	`BOOL`

All these variables are prefixed with `m_` because they are class member variables. This is an MFC naming convention. After the `m_`, a form of Hungarian notation is used, in which the next few letters describe the variable type. In this case, `b` means boolean, and `str` indicates that the variable is a string. You'll see this naming convention in use in this book and other books about programming with Visual C++ and MFC. Following this naming convention will make your code more readable for other programmers; knowing the convention will make it easier for you to read other programmer's code as well.

7. After you add all the necessary variables, click the OK button to close the Class Wizard.

Attaching Functionality to the Controls

Before you begin adding code to all the controls on your application window, you need to add a little bit of code to initialize the variables, setting starting values for most of them. Do this by following these steps:

1. Using the Class Wizard, on the Message Maps tab, select the `OnInitDialog` function in the list of member functions. You can do this by finding the function in the Member Functions list, or by selecting the `CDay2Dlg` object in the list of object IDs and then selecting the `WM_INITDIALOG` message in the messages list, as shown in Figure 2.9.

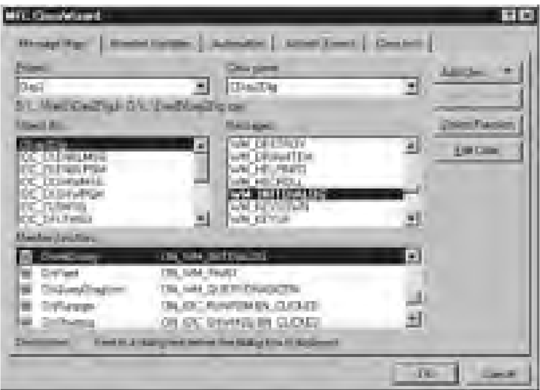


FIGURE 2.9.
You can use the Class Wizard to locate existing functions.

2. Click Edit Code to be taken to the source code for the `OnInitDialog` function.
3. Find the `TODO` marker, which indicates where to begin adding your code, and add the code in Listing 2.1.

LISTING 2.1. DAY2DLG.CPP—THE `OnInitDialog` FUNCTION IS WHERE YOU NEED TO ADD INITIALIZATION CODE.

```
 1: BOOL CDay2Dlg::OnInitDialog()
 2: {
 3:     CDialog::OnInitDialog();
 4:
 5: .
 6: .
 7: .
 8:
 9:     // TODO: Add extra initialization here
10:
11:     ///////////////////////
12:     // MY CODE STARTS HERE
13:     ///////////////////////
14:
15:     // Put a default message in the message edit
16:     m_strMessage = "Place a message here";
17:
18:     // Set all of the check boxes to checked
```

```
19:     m_bShowMsg = TRUE;
20:     m_bShowPgm = TRUE;
21:     m_bEnableMsg = TRUE;
22:     m_bEnablePgm = TRUE;
23:
24:     // Update the dialog with the values
25:     UpdateData(FALSE);
26:
27:     ///////////////////////
28:     // MY CODE ENDS HERE
29:     ///////////////////////
30:
31:     return TRUE;  // return TRUE  unless you set the focus to a
   ➥control
32: }
```

Note

There is more code in the `OnInitDialog` function than has been included in Listing 2.1. I won't include all the code for every function in the code listings throughout this book as a means of focusing on the code that you need to add or modify (and as a means of keeping this book down to a reasonable size). You are welcome to look at the code that has been left out, to learn what it is and what it does, as you build your understanding of MFC and Visual C++.

If you've programmed in C or C++ before, you've noticed that you are setting the value of the `m_strMessage` variable in a very un–C-like manner. It looks more like how you would expect to set a string variable in Visual Basic or PowerBuilder. That's because this variable is a `CString` type variable. The `CString` class enables you to work with strings in a Visual C++ application in much the same way that you would work with strings in one of these other programming languages. However, because this is the C++ programming language, you still need to add a semicolon at the end of each command.

This initialization code is simple. You are setting an initial message in the edit box that you will use to display messages for the user. Next, you are setting all the check boxes to the checked state. It's the last line of the code you added to this function that you really need to notice.

The `UpdateData` function is the key to working with control variables in Visual C++. This function takes the data in the variables and updates the controls on the screen with the variable values. It also takes the data from the controls and populates the attached

variables with any values changed by the user. This process is controlled by the argument passed into the UpdateData function. If the argument is FALSE, the values in the variables are passed to the controls on the window. If the argument is TRUE, the variables are updated with whatever appears in the controls on the window. As a result, which value you pass this function depends on which direction you need to update. After you update one or more variables in your code, then you need to call UpdateData, passing it FALSE as its argument. If you need to read the variables to get their current value, then you need to call UpdateData with a TRUE value before you read any of the variables. You'll get the hang of this as you add more code to your application.

Closing the Application

The first thing that you want to take care of is making sure that the user can close your application. Because you deleted the OK and Cancel buttons and added a new button for closing the application window, you need to place code into the function called by the Exit button to close the window. To do this, follow these steps:

1. Using the Class Wizard, add a function for the IDC_EXIT object on the BN_CLICKED message, as you learned to do yesterday.
2. Click the Edit Code button to take you to the new function that you just added.
3. Enter the code in Listing 2.2.

LISTING 2.2. DAY2DLG.CPP—THE OnExit FUNCTION.

```
 1: void CDay2Dlg::OnExit()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Exit the program
10:     OnOK();
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
```

A single function call within the OnExit function closes the Window and exits the application. Where did this OnOK function come from, and why didn't you have to call it in yesterday's application? Two functions, OnOK and OnCancel, are built into the ancestor

CDialog class from which your CDay2Dlg class is inherited. In the CDialog class, the message map already has the object IDs of the OK and Cancel buttons attached to the OnOK and OnCancel buttons so that buttons with these IDs automatically call these functions. If you had specified the Exit button's object ID as IDOK, you would not have needed to add any code to the button unless you wanted to override the base OnOK functionality.

Showing the User's Message

Showing the message that the user typed into the edit box should be easy because it's similar to what you did in yesterday's application. You can add a function to the Show Message button and call the MessageBox function, as in Listing 2.3.

LISTING 2.3. DAY2DLG.CPP—THE OnShwmsg FUNCTION DISPLAYS THE USER MESSAGE.

```
 1: void CDay2Dlg::OnShwmsg()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Display the message for the user
10:     MessageBox(m_strMessage);
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
```

If you compile and run the application at this point, you'll see one problem with this code. It displays the string that you initialized the m_strMessage variable within the OnInitDialog function. It doesn't display what you type into the edit box. This happens because the variable hasn't been updated with the contents of the control on the window yet. You need to call UpdateData, passing it a TRUE value, to take the values of the controls and update the variables before calling the MessageBox function. Alter the OnShwmsg function as in Listing 2.4.

LISTING 2.4. DAY2DLG.CPP—UPDATED OnShwmsg FUNCTION.

```
 1: void CDay2Dlg::OnShwmsg()
 2: {
```

continues

LISTING 2.4. CONTINUED

```
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Update the message variable with what the user entered
10:     UpdateData(TRUE);
11:
12:     // Display the message for the user
13:     MessageBox(m_strMessage);
14:
15:     ///////////////////////
16:     // MY CODE ENDS HERE
17:     ///////////////////////
18: }
```

Now if you compile and run your application, you should be able to display the message you type into the edit box, as shown in Figure 2.10.

FIGURE 2.10.
The message entered in the edit box is displayed to the user.

Clearing the User's Message

If the user prefers the edit box to be cleared before he or she types a message, you can attach a function to the Clear Message button to clear the contents. You can add this function through the Class Wizard in the usual way. The functionality is a simple matter of setting the `m_strMessage` variable to an empty string and then updating the controls on the window to reflect this. The code to do this is in Listing 2.5.

LISTING 2.5. DAY2DLG.CPP—THE `OnClrmsg` FUNCTION.

```
 1: void CDay2Dlg::OnClrmsg()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Clear the message
```

```
10:     m_strMessage = "";
11:
12:     // Update the screen
13:     UpdateData(FALSE);
14:
15:     ///////////////////////
16:     // MY CODE ENDS HERE
17:     ///////////////////////
18: }
```


Disabling and Hiding the Message Controls

The last thing that you want to do with the message controls is add functionality to the Enable Message Action and Show Message Action check boxes. The first of these check boxes enables or disables the controls dealing with displaying the user message. When the check box is in a checked state, the controls are all enabled. When the check box is in an unchecked state, all those same controls are disabled. In a likewise fashion, the second check box shows and hides this same set of controls. The code for these two functions is in Listing 2.6.

LISTING 2.6. DAY2DLG.CPP—THE FUNCTIONS FOR THE ENABLE AND SHOW MESSAGE ACTIONS CHECK BOXES.

```
 1: void CDay2Dlg::OnCkenblmsg()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the current values from the screen
10:     UpdateData(TRUE);
11:
12:     // Is the Enable Message Action check box checked?
13:     if (m_bEnableMsg == TRUE)
14:     {
15:         // Yes, so enable all controls that have anything
16:         // to do with showing the user message
17:         GetDlgItem(IDC_MSG)->EnableWindow(TRUE);
18:         GetDlgItem(IDC_SHWMSG)->EnableWindow(TRUE);
19:         GetDlgItem(IDC_DFLTMSG)->EnableWindow(TRUE);
20:         GetDlgItem(IDC_CLRMSG)->EnableWindow(TRUE);
21:         GetDlgItem(IDC_STATICMSG)->EnableWindow(TRUE);
22:     }
```

continues

LISTING 2.6. CONTINUED

```
23:     else
24:     {
25:         // No, so disable all controls that have anything
26:         // to do with showing the user message
27:         GetDlgItem(IDC_MSG)->EnableWindow(FALSE);
28:         GetDlgItem(IDC_SHWMSG)->EnableWindow(FALSE);
29:         GetDlgItem(IDC_DFLTMSG)->EnableWindow(FALSE);
30:         GetDlgItem(IDC_CLRMSG)->EnableWindow(FALSE);
31:         GetDlgItem(IDC_STATICMSG)->EnableWindow(FALSE);
32:     }
33:
34:     ///////////////////////
35:     // MY CODE ENDS HERE
36:     ///////////////////////
37: }
38:
39: void CDay2Dlg::OnCkshwmsg()
40: {
41:     // TODO: Add your control notification handler code here
42:
43:     ///////////////////////
44:     // MY CODE STARTS HERE
45:     ///////////////////////
46:
47:     // Get the current values from the screen
48:     UpdateData(TRUE);
49:
50:     // Is the Show Message Action check box checked?
51:     if (m_bShowMsg == TRUE)
52:     {
53:         // Yes, so show all controls that have anything
54:         // to do with showing the user message
55:         GetDlgItem(IDC_MSG)->ShowWindow(TRUE);
56:         GetDlgItem(IDC_SHWMSG)->ShowWindow(TRUE);
57:         GetDlgItem(IDC_DFLTMSG)->ShowWindow(TRUE);
58:         GetDlgItem(IDC_CLRMSG)->ShowWindow(TRUE);
59:         GetDlgItem(IDC_STATICMSG)->ShowWindow(TRUE);
60:     }
61:     else
62:     {
63:         // No, so hide all controls that have anything
64:         // to do with showing the user message
65:         GetDlgItem(IDC_MSG)->ShowWindow(FALSE);
66:         GetDlgItem(IDC_SHWMSG)->ShowWindow(FALSE);
67:         GetDlgItem(IDC_DFLTMSG)->ShowWindow(FALSE);
68:         GetDlgItem(IDC_CLRMSG)->ShowWindow(FALSE);
69:         GetDlgItem(IDC_STATICMSG)->ShowWindow(FALSE);
70:     }
71:
```

```
72:     ////////////////////////
73:     // MY CODE ENDS HERE
74:     ////////////////////////
75: }
```

By now, you should understand the first part of these functions. First, you update the variables with the current values of the controls on the window. Next, you check the value of the boolean variable attached to the appropriate check box. If the variable is `TRUE`, you want to enable or show the control. If the variable if `FALSE`, you want to disable or hide the control.

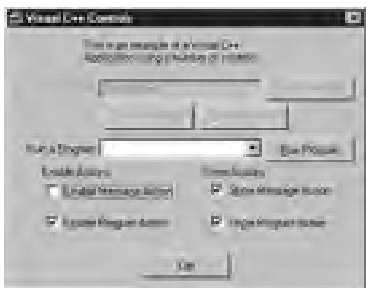
At this point, the code begins to be harder to understand. The first function, GetDlgItem, is passed the ID of the control that you want to change. This function returns the object for that control. You can call this function to retrieve the object for any of the controls on the window while your application is running. The next part of each command is where a member function of the control object is called. The second function is a member function of the object returned by the first function. If you are not clear on how this works, then you might want to check out Appendix A, "C++ Review," to brush up on your C++.

The second functions in these calls, `EnableWindow` and `ShowWindow`, look like they should be used on windows, not controls. Well, yes, they should be used on windows; they happen to be members of the `CWnd` class, which is an ancestor of the `CDialog` class from which your `CDay2Dlg` class is inherited. It just so happens that, in Windows, all controls are themselves windows, completely separate from the window on which they are placed. This allows you to treat controls as windows and to call windows functions on them. In fact, all the control classes are inherited from the `CWnd` class, revealing their true nature as windows.

If you compile and run your application now, you can try the Enable and Show Message Action check boxes. They should work just fine, as shown in Figure 2.11.

FIGURE 2.11.

The user message controls can now be disabled.



Running Another Application

The last major piece of functionality to be implemented in your application is for the set of controls for running another program. If you remember, you added the names of three Windows applications into the combo box, and when you run your application, you can see these application names in the drop-down list. You can select any one of them, and the value area on the combo box is updated with that application name. With that part working as it should, you only need to add code to the Run Program button to actually get the value for the combo box and run the appropriate program. Once you create the function for the Run Program button using the Class Wizard, add the code in Listing 2.7 to the function.

LISTING 2.7. DAY2DLG.CPP—THE OnRunpgm FUNCTION STARTS OTHER WINDOWS APPLICATIONS.

```
 1: void CDay2Dlg::OnRunpgm()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the current values from the screen
10:     UpdateData(TRUE);
11:
12:     // Declare a local variable for holding the program name
13:     CString strPgmName;
14:
15:     // Copy the program name to the local variable
16:     strPgmName = m_strProgToRun;
17:
18:     // Make the program name all uppercase
19:     strPgmName.MakeUpper();
20:
21:     // Did the user select to run the Paint program?
22:     if (strPgmName == "PAINT")
23:         // Yes, run the Paint program
24:         WinExec("pbrush.exe", SW_SHOW);
25:
26:     // Did the user select to run the Notepad program?
27:     if (strPgmName == "NOTEPAD")
28:         // Yes, run the Notepad program
29:         WinExec("notepad.exe", SW_SHOW);
30:
31:     // Did the user select to run the Solitaire program?
32:     if (strPgmName == "SOLITAIRE")
33:         // Yes, run the Solitaire program
34:         WinExec("sol.exe", SW_SHOW);
```

```
35:
36:      ///////////////////////
37:      // MY CODE ENDS HERE
38:      ///////////////////////
39: }
```


As you expect, the first thing that you do in this function is call `UpdateData` to populate the variables with the values of the controls on the window. The next thing that you do, however, might seem a little pointless. You declare a new `CString` variable and copy the value of the combo box to it. Is this really necessary when the value is already in a `CString` variable? Well, it depends on how you want your application to behave. The next line in the code is a call to the `CString` function `MakeUpper`, which converts the string to all uppercase. If you use the `CString` variable that is attached to the combo box, the next time that `UpdateData` is called with `FALSE` as the argument, the value in the combo box is converted to uppercase. Considering that this is likely to happen at an odd time, this is probably not desirable behavior. That's why you use an additional `CString` in this function.

Once you convert the string to all uppercase, you have a series of `if` statements that compare the string to the names of the various programs. When a match is found, the `WinExec` function is called to run the application. Now, if you compile and run your application, you can select one of the applications in the drop-down list and run it by clicking the Run Program button.

It is important to understand the difference in C and C++ between using a single equal sign (=) and a double equal sign (==). The single equal sign performs an assignment of the value on the right side of the equal sign to the variable on the left side of the equal sign. If a constant is on the left side of the equal sign, your program will not compile, and you'll get a nice error message telling you that you cannot assign the value on the right to the constant on the left. The double equal sign (==) is used for comparison. It is important to make certain that you use the double equal sign when you want to compare two values because if you use a single equal sign, you alter the value of the variable on the left. This confusion is one of the biggest sources of logic bugs in C/C++ programs.

The `WinExec` function is an obsolete Windows function. You really should use the `CreateProcess` function instead. However, the `CreateProcess` function has a number of arguments that are difficult to understand this early in programming using Visual C++. The `WinExec` function is still available and is implemented as a macro that calls the `CreateProcess` function. This allows you to use the much simpler `WinExec` function to run another application while still using the function that Windows wants you to use.

Another API function that can be used to run another application is the `ShellExecute` function. This function was originally intended for opening or printing files, but can also be used to run other programs.

Summary

Today, you learned how you can use standard windows controls in a Visual C++ application. You learned how to declare and attach variables to each of these controls and how to synchronize the values between the controls and the variables. You also learned how you can manipulate the controls by retrieving the control objects using their object ID and how you can manipulate the control by treating it as a window. You also learned how to specify the tab order of the controls on your application windows, thus enabling you to control how users navigate your application windows. Finally, you learned how to attach application functionality to the controls on your application window, triggering various actions when the user interacts with various controls. As an added bonus, you learned how you can run other Windows applications from your own application.

Q&A

Q When I specified the object IDs of the controls on the window, three controls had the same ID, `IDC_STATIC`. These controls were the text at the top of the window and the two group boxes. The other two static text controls started out with this same ID until I changed them. How can these controls have the same ID, and why did I have to change the ID on the two static texts where I did change them?

A All controls that don't normally have any user interaction, such as static text and group boxes, are by default given the same object ID. This works fine as long as your application doesn't need to perform any actions on any of these controls. If you do need to interact with one of these controls, as you did with the static text prompts for the edit box and combo box, then you need to give that control a unique ID. In this case, you needed the unique ID to be able to retrieve the control object so that you could enable or disable and show or hide the control. You also

need to assign it a unique ID if you want to attach a variable to the control so that you could dynamically alter the text on the control.

The application behaves in a somewhat unpredictable way if you try to alter any of the static controls that share the same ID. As a general rule of thumb, you can allow static controls to share the same object ID if you are not going to alter the controls at all. If you might need to perform any interaction with the controls, then you need to assign each one a unique object ID.

Q Is there any other way to manipulate the controls, other than retrieving the control objects using their object IDs?

A You can declare variables in the Control category. This basically gives you an object that is the control's MFC class, providing you with a direct way of altering and interacting with the control. You can then call all of the `CWnd` class functions on the control, as you did to enable or disable and show or hide the controls in your application, or you can call the control class methods, enabling you to do things in the code that are specific to that type of control. For instance, if you add another variable to the combo box control and specify that it is a Control category variable, you can use it to add items to the drop-down list on the control.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises appear in Appendix B, "Answers."

Quiz

1. Why do you need to specify the tab order of the controls on your application windows?
2. How can you include a mnemonic in a static text field that will take the user to the edit box or combo box beside the text control?
3. Why do you need to give unique object IDs to the static text fields in front of the edit box and combo boxes?
4. Why do you need to call the `UpdateData` function before checking the value of one of the controls?

Exercises

1. Add code to the Default Message button to reset the edit box to say `Enter a message here`.

2. Add code to enable or disable and show or hide the controls used to select and run another application.
3. Extend the code in the `OnRunpgm` function to allow the user to enter his own program name to be run.

WEEK 1

DAY 3

Allowing User Interaction—Integrating the Mouse and Keyboard in Your Application

Depending on the type of application you are creating, you might need to notice what the user is doing with the mouse. You need to know when and where the mouse was clicked, which button was clicked, and when the button was released. You also need to know what the user did while the mouse button was being held down.

Another thing that you might need to do is read the keyboard events. As with the mouse, you might need to know when a key was pressed, how long it was held down, and when it was released.

Today you are going to learn

- What mouse events are available for use and how to determine which one is appropriate for your application's needs.
- How you can listen to mouse events and how to react to them in your Visual C++ application.
- What keyboard events are available for use and what actions will trigger each of these events.
- How to capture keyboard events and take action based on what the user pressed.

Understanding Mouse Events

As you learned yesterday, when you are working with most controls, you are limited to a select number of events that are available in the Class Wizard. When it comes to mouse events, you are limited for the most part to click and double-click events. Just looking at your mouse tells you that there must be more to capturing mouse events than recognizing these two. What about the right mouse button? How can you tell if it has been pressed? And what about drawing programs? How can they follow where you drag the mouse?

If you open the Class Wizard in one of your projects, select the dialog in the list of object IDs, and then scroll through the list of messages that are available, you will find a number of mouse-related events, which are also listed in Table 3.1. These event messages enable you to perform any task that might be required by your application.

TABLE 3.1. MOUSE EVENT MESSAGES.

Message	*Description*
`WM_LBUTTONDOWN`	The left mouse button has been pressed.
`WM_LBUTTONUP`	The left mouse button has been released.
`WM_LBUTTONDBLCLK`	The left mouse button has been double-clicked.
`WM_RBUTTONDOWN`	The right mouse button has been pressed.
`WM_RBUTTONUP`	The right mouse button has been released.
`WM_RBUTTONDBLCLK`	The right mouse button has been double-clicked.
`WM_MOUSEMOVE`	The mouse is being moved across the application window space.
`WM_MOUSEWHEEL`	The mouse wheel is being moved.

Drawing with the Mouse

Today you are going to build a simple drawing program that uses some of the available mouse events to let the user draw simple figures on a dialog window. This application depends mostly on the `WM_MOUSEMOVE` event message, which signals that the mouse is being moved. You will look at how you can tell within this event function whether the left mouse button is down or up. You will also learn how you can tell where the mouse is on the window. Sound's fairly straight ahead, so let's get going by following these steps:

1. Create a new MFC AppWizard workspace project, calling the project **`Mouse`**.
2. Specify that this project will be a dialog-based application in the first AppWizard step.
3. Use the default settings in the AppWizard. In the second step, specify a suitable dialog title, such as **`Mouse and Keyboard`**.
4. After the application shell is created, remove all controls from the dialog window. This provides the entire dialog window surface for drawing. This step is also necessary for your application to capture any keyboard events.

If there are any controls on a dialog, all keyboard events are directed to the control that currently has input focus—the control that is highlighted or has the cursor visible in it. To capture any keyboard events in a dialog, you have to remove all controls from the dialog.

5. Open the Class Wizard. Select `WM_MOUSEMOVE` from the list of messages, and add a function by clicking the Add Function button. Click the OK button to accept the suggested function name.
6. Click the Edit Code button to edit the `OnMouseMove` function you just created, adding the code in Listing 3.1.

LISTING 3.1. THE `OnMouseMove` FUNCTION.

```
1: void CMouseDlg::OnMouseMove(UINT nFlags, CPoint point)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     ///////////////////////
6:     // MY CODE STARTS HERE
7:     ///////////////////////
8:
```

continues

LISTING 3.1. CONTINUED

```
 9:     // Check to see if the left mouse button is down
10:     if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
11:     {
12:         // Get the Device Context
13:         CClientDC dc(this);
14:
15:         // Draw the pixel
16:         dc.SetPixel(point.x, point.y, RGB(0, 0, 0));
17:     }
18:
19:     ///////////////////////
20:     // MY CODE ENDS HERE
21:     ///////////////////////
22:
23:     CDialog::OnMouseMove(nFlags, point);
24: }
```

Look at the function definition at the top of the listing. You will notice that two arguments are passed into this function. The first of these arguments is a set of flags that can be used to determine whether a mouse button is depressed (and which one). This determination is made in the first line of your code with the `if` statement:

```
if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
```

In the first half of the condition being evaluated, the flags are filtered down to the one that indicates that the left mouse button is down. In the second half, the filtered flags are compared to the flag that indicates that the left mouse button is down. If the two match, then the left mouse button is down.

The second argument to this function is the location of the mouse. This argument gives you the coordinates on the screen where the mouse currently is. You can use this information to draw a spot on the dialog window.

Before you can draw any spots on the dialog window, you need to get the device context for the dialog window. This is done by declaring a new instance of the `CClientDC` class. This class encapsulates the device context and most of the operations that can be performed on it, including all the screen drawing operations. In a sense, the device context is the canvas upon which you can draw with your application. Until you have a canvas, you cannot do any drawing or painting. After the device context object is created, you can call its `SetPixel` function, which colors the pixel at the location specified in the first two arguments with the color specified in the third argument. If you compile and run your program, you can see how it allows you to draw on the window surface with the mouse, as shown in Figure 3.1.

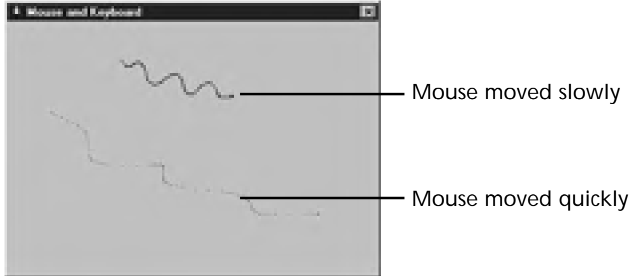


FIGURE 3.1.
Drawing on the window with the mouse.

In Windows, colors are specified as a single number that is a combination of three numbers. The three numbers are the brightness levels for the red, green, and blue pixels in your computer display. The `RGB` function in your code is a macro that combines these three separate values into the single number that must be passed to the `SetPixel` function or to any other function that requires a color value. These three numbers can be any value between and including 0 and 255.

3

Using the AND and OR Binaries

If you are new to C++, you need to understand how the different types of `AND` and `OR` work. The two categories of `AND`s and `OR`s are logical and binary. The logical `AND`s and `OR`s are used in logical or conditional statements, such as an `if` or `while` statement that is controlling the logic flow. The binary `AND`s and `OR`s are used to combine two values on a binary level.

The ampersand character (`&`) is used to denote `AND`. A single ampersand (`&`) is a binary `AND`, and a double ampersand (`&&`) is a logical `AND`. A logical `AND` works much like the word `AND` in Visual Basic or PowerBuilder. It can be used in an `if` statement to say "if this condition AND this other condition…" where both conditions must be true before the entire statement is true. A binary `AND` is used to set or unset bits. When two values are binary `AND`ed, only the bits that are set to 1 in both values remain as 1; all the rest of the bits are set to 0. To understand how this works, start with two 8-bit values such as the following:

Value 1	01011001
Value 2	00101001

If you binary `AND` these two values together, you wind up with the following value:

`AND`ed Value	00001001

All the bits that had 1 in one of the values, but not in the other value, were set to 0. All the bits that were 1 in both values remained set to 1. All the bits that were 0 in both values remained 0.

OR is represented by the pipe character (¦), and as with AND, a single pipe (¦) is a binary OR, whereas a double pipe (¦¦) is a logical OR. As with AND, a logical OR can be used in conditional statements such as `if` or `while` statements to control the logical flow, much like the word OR in Visual Basic and PowerBuilder. It can be used in an `if` statement to say "if this condition OR this other condition…" and if either condition is true, the entire statement is true. You can use a binary OR to combine values on a binary level. With OR, if a bit is set to 1 in either value, the resulting bit is set to 1. With a binary OR, the only way that a bit is set to 0 in the resulting value is if the bit was already 0 in both values. Take the same two values that were used to illustrate the binary AND:

Value 1	01011001
Value 2	00101001

If you binary OR these two values together, you get the following value:

ORed Value	01111001

In this case, every bit that was set to 1 in either value was set to 1 in the resulting value. Only those bits that were 0 in both values were 0 in the resulting value.

Binary Attribute Flags

Binary ANDs and ORs are used in C++ for setting and reading attribute flags. Attribute flags are values where each bit in the value specifies whether a specific option is turned on or off. This enables programmers to use defined flags. A defined flag is a value with only one bit set to 1 or a combination of other values in which a specific combination of bits is set to 1 so that multiple options are set with a single value. The flags controlling various options are ORed together, making a composite flag specifying which options should be on and which should be off.

If two flags that specify certain conditions are specified as two different bits in a byte, those two flags can often be ORed together as follows:

Flag 1	00001000
Flag 2	00100000
Combination	**00101000**

This is how flags are combined to specify a number of settings in a limited amount of memory space. In fact, this is what is done with most of the check box settings on the

window and control properties dialogs. These on/off settings are ORed together to form one or two sets of flags that are examined by the Windows operating system to determine how to display the window or control and how it should behave.

On the flip side of this process, when you need to determine if a specific flag is included in the combination, you can AND the combination flag with the specific flag that you are looking for as follows:

Combination	00101000
Flag 1	00001000
Result	**00001000**

The result of this operation can be compared to the flag that you used to filter the combined flag. If the result is the same, the flag was included. Another common approach is to check whether the filtered combination flag is nonzero. If the flag being used for filtering the combination had not been included, the resulting flag would be zero. As a result, you could have left the comparison out of the if statement in the preceding code, leaving you with an if statement that looks like the following:

```
if (nFlags & MK_LBUTTON)
```

You can modify this approach to check whether a flag is not in the combination as follows:

```
if (!(nFlags & MK_LBUTTON))
```

You might find one of these ways of checking for a flag easier to understand than the others. You'll probably find all of them in use.

Improving the Drawing Program

If you ran your program, you probably noticed a small problem. To draw a solid line, you need to move the mouse very slowly. How do other painting programs solve this problem? Simple, they draw a line between two points drawn by the mouse. Although this seems a little like cheating, it's the way that computer drawing programs work.

As you move the mouse across the screen, your computer is checking the location of the mouse every few clock ticks. Because your computer doesn't have a constant trail of where your mouse has gone, it has to make some assumptions. The way your computer makes these assumptions is by taking the points that the computer does know about and drawing lines between them. When you draw lines with the freehand tool in Paint, your computer is playing connect the dots.

Because all the major drawing programs draw lines between each pair of points, what do you need to do to adapt your application so that it also uses this technique? First, you need to keep track of the previous position of the mouse. This means you need to add two variables to the dialog window to maintain the previous X and Y coordinates. You can do this by following these steps:

1. In the workspace pane, select the Class View tab.
2. Select the dialog class—in this case, the `CMouseDlg` class.
3. Right-click the mouse and select Add Member Variable from the pop-up menu.
4. Enter **`int`** as the Variable Type and **`m_iPrevY`** as the Variable Name and specify Private for the access in the Add Member Variable dialog, as shown in Figure 3.2.

FIGURE 3.2.
The Add Member Variable dialog.

5. Click OK to add the variable.
6. Repeat steps 3 through 5, specifying the Variable Name as **`m_iPrevX`** to add the second variable.

After you add the variables needed to keep track of the previous mouse position, you can make the necessary modifications to the `OnMouseMove` function, as shown in Listing 3.2.

LISTING 3.2. THE REVISED `OnMouseMove` FUNCTION.

```
 1: void CMouseDlg::OnMouseMove(UINT nFlags, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Check to see if the left mouse button is down
10:     if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
11:     {
12:         // Get the Device Context
13:         CClientDC dc(this);
14:
15:         // Draw a line from the previous point to the current point
16:         dc.MoveTo(m_iPrevX, m_iPrevY);
```

```
17:         dc.LineTo(point.x, point.y);
18:
19:         // Save the current point as the previous point
20:         m_iPrevX = point.x;
21:         m_iPrevY = point.y;
22:     }
23:
24:     ///////////////////////
25:     // MY CODE ENDS HERE
26:     ///////////////////////
27:
28:     CDialog::OnMouseMove(nFlags, point);
29: }
```


Look at the code that draws the line from the previous point to the current point:

```
dc.MoveTo(m_iPrevX, m_iPrevY);
dc.LineTo(point.x, point.y);
```

You see that you need to move to the first position and then draw a line to the second point. The first step is important because without it, there is no telling where Windows might think the starting position is. If you compile and run your application, it draws a bit better. However, it now has a peculiar behavior. Every time you press the left mouse button to begin drawing some more, your application draws a line from where you ended the last line you drew, as shown in Figure 3.3.

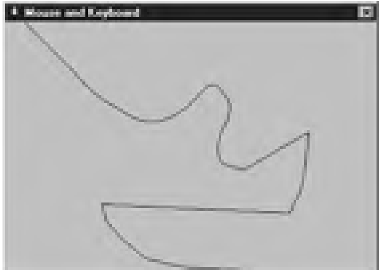


FIGURE 3.3.
The drawing program with a peculiar behavior.

Adding the Finishing Touches

Your application is doing all its drawing on the mouse move event when the left button is held down. Initializing the previous position variables with the position of the mouse when the left button is pressed should correct this application behavior. Let's try this approach by following these steps:

1. Using the Class Wizard, add a function for the `WM_LBUTTONDOWN` message on the dialog object.

2. Edit the `OnLButtonDown` function that you just created, adding the code in Listing 3.3.

LISTING 3.3. THE `OnLButtonDown` FUNCTION.

```
 1: void CMouseDlg::OnLButtonDown(UINT nFlags, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Set the current point as the starting point
10:     m_iPrevX = point.x;
11:     m_iPrevY = point.y;
12:
13:     ///////////////////////
14:     // MY CODE ENDS HERE
15:     ///////////////////////
16:
17:     CDialog::OnLButtonDown(nFlags, point);
18: }
```

When you compile and run your application, you should find that you can draw much like you would expect with a drawing program, as shown in Figure 3.4.

FIGURE 3.4.
The finished drawing program.

Capturing Keyboard Events

Reading keyboard events is similar to reading mouse events. As with the mouse, there are event messages for when a key is pressed and when it is released. These events are listed in Table 3.2.

TABLE 3.2. KEYBOARD EVENT MESSAGES.

Message	*Description*
`WM_KEYDOWN`	A key has been pressed down.
`WM_KEYUP`	A key has been released.

The keyboard obviously has fewer messages than the mouse does. Then again, there are only so many things that you can do with the keyboard. These event messages are available on the dialog window object and are triggered only if there are no enabled controls on the window. Any enabled controls on the window have input focus, so all keyboard events go to them. That's why you remove all controls from the main dialog for your drawing application.

Changing the Drawing Cursor

To get a good idea of how you can use keyboard-related event messages, why don't you use certain keys to change the mouse cursor in your drawing application? Make the A key change the cursor to the default arrow cursor, which your application starts with. Then you can make B change the cursor to the I-beam and C change the cursor to the hourglass. To get started adding this functionality, follow these steps:

1. Using the Class Wizard, add a function for the `WM_KEYDOWN` message on the dialog object.
2. Edit the `OnKeyDown` function that you just created, adding the code in Listing 3.4.

LISTING 3.4. THE `OnKeyDown` FUNCTION.

```
 1: void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     char lsChar;        // The current character being pressed
10:     HCURSOR lhCursor;    // The handle to the cursor to be displayed
11:
12:     // Convert the key pressed to a character
13:     lsChar = char(nChar);
14:
15:     // Is the character "A"
16:     if (lsChar == 'A')
```

continues

LISTING 3.4. CONTINUED

```
17:     {
18:         // Load the arrow cursor
19:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
20:         // Set the screen cursor
21:         SetCursor(lhCursor);
22:     }
23:
24:     // Is the character "B"
25:     if (lsChar == 'B')
26:     {
27:         // Load the I beam cursor
28:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_IBEAM);
29:         // Set the screen cursor
30:         SetCursor(lhCursor);
31:     }
32:
33:     // Is the character "C"
34:     if (lsChar == 'C')
35:     {
36:         // Load the hourglass cursor
37:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_WAIT);
38:         // Set the screen cursor
39:         SetCursor(lhCursor);
40:     }
41:
42:     // Is the character "X"
43:     if (lsChar == 'X')
44:     {
45:         // Load the arrow cursor
46:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
47:         // Set the screen cursor
48:         SetCursor(lhCursor);
49:         // Exit the application
50:         OnOK();
51:     }
52:
53:     ///////////////////////
54:     // MY CODE ENDS HERE
55:     ///////////////////////
56:
57:     CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
58: }
```

In the function definition, you see three arguments to the `OnKeyDown` function. The first is the key that was pressed. This argument is the character code of the character, which needs to be converted into a character in the first line of your code. After you convert the

character, you can perform straight-ahead comparisons to determine which key was pressed:

```
void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
```

The second argument to the `OnKeyDown` function is the number of times that the key is pressed. Normally, if the key is pressed and then released, this value is 1. If the key is pressed and held down, however, the repeat count rises for this key. In the end, this value tells you how many times that Windows thinks the key has been pressed.

The third argument to the `OnKeyDown` function is a combination flag that can be examined to determine whether the Alt key was pressed at the same time as the key or whether the key being pressed is an extended key. This argument does not tell you whether the shift or control keys were pressed.

When you determine that a specific key was pressed, then it's time to change the cursor to whichever cursor is associated with that key. There are two steps to this process. The first step is to load the cursor into memory. You accomplish this step with the `LoadStandardCursor` function, which loads one of the standard Windows cursors and returns a handle to the cursor.

A sister function, `LoadCursor`, can be passed the file or resource name of a custom cursor so that you can create and load your own cursors. If you design your own cursor in the resource editor in Visual C++, you can pass the cursor name as the only argument to the `LoadCursor` function. For example, if you create your own cursor and name it `IDC_MYCURSOR`, you can load it with the following line of code:

```
lhCursor = AfxGetApp()->LoadCursor(IDC_MYCURSOR);
```

After you load your own cursor, you can set the mouse pointer to your cursor using the `SetCursor` function, as with a standard cursor.

After the cursor is loaded into memory, the handle to that cursor is passed to the `SetCursor` function, which switches the cursor to the one the handle points to. If you compile and run your application, you should be able to press one of these keys and get the cursor to change, as in Figure 3.5. However, the moment you move the mouse to do any drawing, the cursor switches back to the default arrow cursor. The following section describes how to make your change stick.

FIGURE 3.5.
Changing the cursor with specific keys.

Making the Change Stick

The problem with your drawing program is that the cursor is redrawn every time you move the mouse. There must be some way of turning off this behavior.

Each time the cursor needs to be redrawn—because the mouse has moved, because another window that was in front of your application has gone away, or because of whatever other reason—a `WM_SETCURSOR` event message is sent to your application. If you override the native behavior of your application on this event, the cursor you set remains unchanged until you change it again. To do this, follow these steps:

1. Add a new variable to the `CMouseDlg` class, as you did for the previous position variables. This time, declare the type as `BOOL` and name the variable **`m_bCursor`**, as shown in Figure 3.6.

FIGURE 3.6.
Defining a class member variable.

2. Initialize the `m_bCursor` variable in the `OnInitDialog` with the code in Listing 3.5.

LISTING 3.5. THE `OnInitDialog` FUNCTION.

```
1: BOOL CMouseDlg::OnInitDialog()
2: {
3:     CDialog::OnInitDialog();
4:
5: .
6: .
7: .
8:     // Set the icon for this dialog.  The framework does this
       ➥automatically
```

```
 9:     //  when the application's main window is not a dialog
10:     SetIcon(m_hIcon, TRUE);            // Set big icon
11:     SetIcon(m_hIcon, FALSE);        // Set small icon
12:
13:     // TODO: Add extra initialization here
14:
15:     ///////////////////////
16:     // MY CODE STARTS HERE
17:     ///////////////////////
18:
19:     // Initialize the cursor to the arrow
20:     m_bCursor = FALSE;
21:
22:     ///////////////////////
23:     // MY CODE ENDS HERE
24:     ///////////////////////
25:
26:     return TRUE;  // return TRUE  unless you set the focus to a
        ➥control
27: }
```


3. Alter the `OnKeyDown` function to set the `m_bCursor` flag to `TRUE` when you change the cursor, as in Listing 3.6.

LISTING 3.6. THE `OnKeyDown` FUNCTION.

```
 1: void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     char lsChar;        // The current character being pressed
10:     HCURSOR lhCursor;    // The handle to the cursor to be displayed
11:
12:     // Convert the key pressed to a character
13:     lsChar = char(nChar);
14:
15:     // Is the character "A"
16:     if (lsChar == 'A')
17:         // Load the arrow cursor
18:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
19:
20:     // Is the character "B"
21:     if (lsChar == 'B')
```

continues

LISTING 3.6. CONTINUED

```
22:         // Load the I beam cursor
23:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_IBEAM);
24:
25:     // Is the character "C"
26:     if (lsChar == 'C')
27:         // Load the hourglass cursor
28:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_WAIT);
29:
30:     // Is the character "X"
31:     if (lsChar == 'X')
32:     {
33:         // Load the arrow cursor
34:         lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
35:         // Set the cursor flag
36:         m_bCursor = TRUE;
37:         // Set the screen cursor
38:         SetCursor(lhCursor);
39:         // Exit the application
40:         OnOK();
41:     }
42:     else
43:     {
44:         // Set the cursor flag
45:         m_bCursor = TRUE;
46:         // Set the screen cursor
47:         SetCursor(lhCursor);
48:     }
49:
50:     ///////////////////////
51:     // MY CODE ENDS HERE
52:     ///////////////////////
53:
54:     CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
55: }
```

4. Using the Class Wizard, add a function for the `WM_SETCURSOR` message on the dialog object.
5. Edit the `OnSetCursor` function that you just created, adding the code in Listing 3.7.

LISTING 3.7. THE `OnSetCursor` FUNCTION.

```
1: BOOL CMouseDlg::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
```

```
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // If the cursor has been set, then return TRUE
10:     if (m_bCursor)
11:         return TRUE;
12:     else
13:
14:     ///////////////////////
15:     // MY CODE ENDS HERE
16:     ///////////////////////
17:
18:     return CDialog::OnSetCursor(pWnd, nHitTest, message);
19: }
```


The `OnSetCursor` function needs to always return `TRUE` or else call the ancestor function. The ancestor function resets the cursor and does need to be called when the application first starts. Because of this, you need to initialize your variable to `FALSE` so that until the user presses a key to change the cursor, the default `OnSetCursor` processing is executed. When the user changes the cursor, you want to bypass the default processing and return `TRUE` instead. This allows the user to draw with whichever cursor has been selected, including the hourglass, as shown in Figure 3.7.

FIGURE 3.7.
Drawing with the hourglass cursor.

The most common cursor change that you are likely to use in your programs is setting the cursor to the hourglass while your program is working on something that might take a while. There are actually two functions available in MFC that you can use to handle this task. The first is `BeginWaitCursor`, which displays the hourglass cursor for the user. The second function is `EndWaitCursor`, which restores the cursor to the default cursor. Both of these functions are members of the `CCmdTarget` class, from which all of the MFC window and control classes are derived.

If you have a single function controlling all the processing during which you need to display the hourglass and you don't need to display the hourglass after the function has finished, an easier way to show the hourglass cursor is to declare a variable of the `CWaitCursor` class at the beginning of the function. This automatically displays the hourglass cursor for the user. As soon as the program exits the function, the cursor will be restored to the previous cursor.

Summary

In this chapter, you learned about how you can capture mouse event messages and perform some simple processing based upon these events. You used the mouse events to build a simple drawing program that you could use to draw freehand figures on a dialog window.

You also learned how to grab keyboard events and determine which key is being pressed. You used this information to determine which cursor to display for drawing. For this to work, you had to learn about the default cursor drawing in MFC applications and how you could integrate your code with this behavior to make your application behave the way you want it to.

From here, you will learn how to use the Windows timer to trigger events at regular intervals. You will also learn how to use additional dialog windows to get feedback from the user so that you can integrate that feedback into how your application behaves. After that, you will learn how to create menus for your applications.

Q&A

Q How can I change the type of line that I am drawing? I would like to draw a larger line with a different color.

A When you use any of the standard device context commands to draw on the screen, you are drawing with what is known as a pen, much like the pen you use to draw on a piece of paper. To draw bigger lines, or different color lines, you need to select a new pen. You can do this by adapting the code in the `OnMouseMove` function, starting where you get the device context. The following code enables you to draw with a big red pen:

```
// Get the Device Context
CClientDC dc(this);

// Create a new pen
CPen lpen(PS_SOLID, 16, RGB(255, 0, 0));
```

```
// Use the new pen
dc.SelectObject(&lpen);

// Draw a line from the previous point to the current point
dc.MoveTo(m_iPrevX, m_iPrevY);
dc.LineTo(point.x, point.y);
```

Q How can you tell whether the Shift or Ctrl keys are being held down when you receive the `WM_KEYDOWN` message?

A You can call another function, `::GetKeyState`, with a specific key code to determine whether that key is being held down. If the return value of the `::GetKeyState` function is negative, the key is being held down. If the return value is nonnegative, the key is not being held down. For instance, if you want to determine whether the Shift key is being held down, you can use this code:

```
if (::GetKeyState(VK_SHIFT) < 0)
   MessageBox("Shift key is down!");
```

A number of virtual key codes are defined in Windows for all the special keys. These codes let you look for special keys without worrying about OEM scan codes or other key sequences. You can use these virtual key codes in the `::GetKeyState` function and pass them to the `OnKeyDown` function as the `nChar` argument. Refer to the Visual C++ documentation for a list of the virtual key codes.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What are the possible mouse messages that you can add functions for?
2. How can you tell if the left mouse button is down on the `WM_MOUSEMOVE` event message?
3. How can you prevent the cursor from changing back to the default cursor after you set it to a different one?

Exercises

1. Modify your drawing program so that the left mouse button can draw in red and the right mouse button can draw in blue.

2. Extend the `OnKeyDown` function to add some of the following standard cursors:
    - `IDC_CROSS`
    - `IDC_UPARROW`
    - `IDC_SIZEALL`
    - `IDC_SIZENWSE`
    - `IDC_SIZENESW`
    - `IDC_SIZEWE`
    - `IDC_SIZENS`
    - `IDC_NO`
    - `IDC_APPSTARTING`
    - `IDC_HELP`

WEEK 1

DAY 4

Working with Timers

You may often find yourself building an application that needs to perform a specific action on a regular basis. The task can be something simple such as displaying the current time in the status bar every second or writing a recovery file every five minutes. Both of these actions are regularly performed by several applications that you probably use on a daily basis. Other actions that you might need to perform include checking specific resources on a regular basis, as a resource monitor or performance monitor does. These examples are just a few of the situations where you want to take advantage of the availability of timers in the Windows operating system.

Today you are going to learn

- How to control and use timers in your Visual C++ applications.
- How to set multiple timers, each with a different recurrence interval.
- How to know which timer has triggered.
- How you can incorporate this important resource into all your Visual C++ applications.

Understanding Windows Timers

Windows timers are mechanisms that let you set one or more timers to be triggered at a specific number of milliseconds. If you set a timer to be triggered at a 1,000 millisecond interval, it triggers every second. When a timer triggers, it sends a `WM_TIMER` message to your application. You can use the Class Wizard to add a function to your application to handle this timer message.

Timer events are placed only in the application event queue if that queue is empty and the application is idle. Windows does not place timer event messages in the application event queue if the application is already busy. If your application has been busy and has missed several timer event messages, Windows places only a single timer message in the event queue. Windows does not send your application all the timer event messages that occurred while your application was busy. It doesn't matter how many timer messages your application may have missed; Windows still places only a single timer message in your queue.

When you start or stop a timer, you specify a timer ID, which can be any integer value. Your application uses this timer ID to determine which timer event has triggered, as well as to start and stop timers. You'll get a better idea of how this process works as you build your application for today.

Placing a Clock on Your Application

In the application that you will build today, you will use two timers. The first timer maintains a clock on the window. This timer is always running while the application is running. The second timer is configurable to trigger at whatever interval the user specifies in the dialog. The user can start and stop this timer at will. Let's get started.

Creating the Project and Application

You will build today's sample application in three phases. In the first phase, you will add all the controls necessary for the entire application. In the second phase, you will add the first of the two timers. This first timer will control the clock on the application dialog. In the third phase, you will add the second timer, which the user can tune, start, and stop as desired.

To create today's application, follow these steps:

1. Create a new project, named **`Timers`**, using the same AppWizard settings that you've used for the past three days. Specify the application title as **`Timers`**.

2. Lay out the dialog window as shown in Figure 4.1, using the control properties in Table 4.1. Remember that when you place a control on the window, you can right-click the mouse to open the control's properties from the pop-up menu.

FIGURE 4.1.
The Timers application dialog layout.

4

TABLE 4.1. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`Timer &Interval:`
Edit Box	ID	`IDC_INTERVAL`
Button	ID	`IDC_STARTTIME`
	Caption	`&Start Timer`
Button	ID	`IDC_STOPTIMER`
	Caption	`S&top Timer`
	Disabled	Checked
Static Text	ID	`IDC_STATIC`
	Caption	`Time:`
Static Text	ID	`IDC_STATICTIME`
	Caption	`Current Time`

continues

TABLE 4.1. CONTINUED

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`Count:`
Static Text	ID	`IDC_STATICCOUNT`
	Caption	`0`
Button	ID	`IDC_EXIT`
	Caption	`E&xit`

3. Set the tab order as you learned on Day 2, "Using Controls in Your Application."
4. Add code to the Exit button to close the application, as you did on Day 2.

Adding the Timer IDs

Because you will be using two timers in this application, you should add two IDs to your application to represent the two timer IDs. This can be done by following these steps:

1. On the Resource View tab in the workspace pane, right-click the mouse over the Timers resources folder at the top of the resource tree. Select Resource Symbols from the pop-up menu, as in Figure 4.2.

FIGURE 4.2.

The Resource pop-up menu.

2. On the Resource Symbols dialog, click the New button.

3. On the New Symbol dialog, enter **ID_CLOCK_TIMER** as the symbol name and **1** as the value, as shown in Figure 4.3.

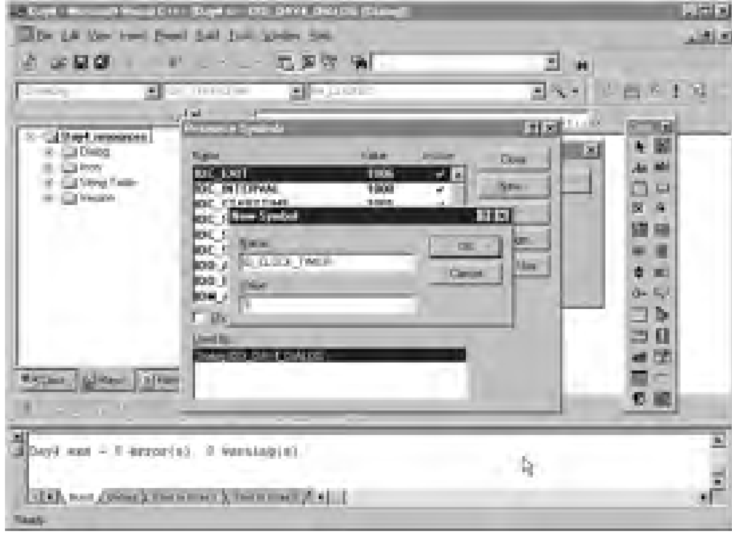


FIGURE 4.3.
Adding a new resource symbol.

4. Repeat steps 2 and 3, specifying **ID_COUNT_TIMER** as the symbol name and **2** as the value.
5. Click the Close button to close the Resource Symbols dialog. The two timer IDs are now in your application and ready for use.

Starting the Clock Timer

To start the clock timer, you need to edit the `OnInitDialog` function, as you did in the previous two days. Add the new code in Listing 4.1.

LISTING 4.1. THE `OnInitDialog` FUNCTION.

```
1: BOOL CTimersDlg::OnInitDialog()
2: {
3:     CDialog::OnInitDialog();
4: .
5: .
6: .
7:     // TODO: Add extra initialization here
8:
9:     ///////////////////////
```

continues

LISTING 4.1. CONTINUED

```
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Start the clock timer
14:     SetTimer(ID_CLOCK_TIMER, 1000, NULL);
15:
16:     ///////////////////////
17:     // MY CODE ENDS HERE
18:     ///////////////////////
19:
20:     return TRUE;  // return TRUE  unless you set the focus to a
                      ➥control
21: }.
```

In this listing, you started the clock timer with the `SetTimer` function. The first argument that you passed to the `SetTimer` function is the ID for the clock timer. The second argument is how often you want to trigger the event. In this case, the clock timer event is triggered every 1,000 milliseconds, or about every second. The third argument is the address of an optional callback function that you can specify to bypass the `WM_TIMER` event. If you pass `NULL` for this argument, the `WM_TIMER` event is placed in the application message queue.

Note

A callback function is a function you create that is called directly by the Windows operating system. Callback functions have specific argument definitions, depending on which subsystem calls the function and why. After you get past the function definition, however, you can do whatever you want or need to do in the function.

A callback function works by passing the address of the function as an argument to a Windows function that accepts callback functions as arguments. When you pass the function address to Windows, your function is called directly every time the circumstances occur that require Windows to call the callback function.

Handling the Clock Timer Event

Now that you've started a timer, you need to add the code to handle the timer event message. You can do this by following these steps:

1. Using the Class Wizard, add a variable to the `IDC_STATICTIME` control of type `CString` named `m_sTime`.

2. Using the Class Wizard, add a function to handle the `WM_TIMER` message for the `CTimersDlg` object.
3. Edit the `OnTimer` function, adding the code in Listing 4.2.

LISTING 4.2. THE `OnTimer` FUNCTION.

```
 1: void CTimersDlg::OnTimer(UINT nIDEvent)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the current time
10:     CTime curTime = CTime::GetCurrentTime();
11:
12:     // Display the current time
13:     m_sTime.Format("%d:%d:%d", curTime.GetHour(),
14:         curTime.GetMinute(),
15:         curTime.GetSecond());
16:
17:     // Update the dialog
18:     UpdateData(FALSE);
19:
20:     ///////////////////////
21:     // MY CODE ENDS HERE
22:     ///////////////////////
23:
24:     CDialog::OnTimer(nIDEvent);
25: }
```

In this listing, you declare an instance of the `CTime` class, initializing it to the current system time. The next thing that you do is set the `m_sTime` string to the current time, using the `Format` method to format the time in the familiar HH:MM:SS format. Finally, you update the dialog window with the current time. If you compile and run your application now, you should see a clock running in the middle of your dialog window, as in Figure 4.4.

FIGURE 4.4.

A running clock on your application dialog.

4

Adding a Second Timer to Your Application

As you have seen, adding a single timer to an application is a pretty simple task. All it takes is calling the `SetTimer` function and then placing the timer code in the `OnTimer` function. However, sometimes you need more than one timer running simultaneously in the same application. Then things get a little bit more involved.

Adding the Application Variables

Before you add the second timer to your application, you need to add a few variables to the controls. With the clock timer, you needed only a single variable for updating the clock display. Now you need to add a few other variables for the other controls, as listed in Table 4.2.

TABLE 4.2. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
IDC_STATICCOUNT	m_sCount	Value	CString
IDC_INTERVAL	m_iInterval	Value	int
IDC_STARTTIME	m_cStartTime	Control	CButton
IDC_STOPTIMER	m_cStopTime	Control	CButton

After you add all the variables using the Class Wizard, follow these steps:

1. Using the Class Wizard, select the `m_iInterval` variable and specify a Minimum Value of `1` and a Maximum Value of `100000` in the two edit boxes below the list of variables, as shown in Figure 4.5.

FIGURE 4.5.
Specifying a range for a variable.

2. On the Class View tab in the workspace pane, add a member variable to the `CTimersDlg` class as you learned yesterday. Specify the variable type as `int`, the variable name as `m_iCount`, and the access as `Private`.
3. Using the Class Wizard, add a function on the `EN_CHANGE` event message for the `IDC_INTERVAL` control ID (the edit box). Edit the function and add the code in Listing 4.3.

LISTING 4.3. THE `OnChangeInterval` FUNCTION.

```
 1: void CTimersDlg::OnChangeInterval()
 2: {
 3:     // TODO: If this is a RICHEDIT control, the control will not
 4:     // send this notification unless you override the
        ➥CDialog::OnInitialUpdate()
 5:     // function and call CRichEditCrtl().SetEventMask()
 6:     // with the EN_CHANGE flag ORed into the mask.
 7:
 8:     // TODO: Add your control notification handler code here
 9:
10:     ///////////////////////
11:     // MY CODE STARTS HERE
12:     ///////////////////////
13:
14:     // Update the variables
15:     UpdateData(TRUE);
16:
17:     ///////////////////////
18:     // MY CODE ENDS HERE
19:     ///////////////////////
20: }
```

When you specify a value range for the timer interval variable, Visual C++ automatically prompts the user, stating the available value range if the user enters a value outside of the specified range. This prompt is triggered by the `UpdateData` function call in the `OnChangeInterval` function. The last variable that was added through the workspace pane is used as the actual counter, which is incremented with each timer event.

Starting and Stopping the Counting Timer

To make your second timer operational, you need to

- Initialize the `m_iInterval` variable.
- Start the timer when the `IDC_STARTTIME` button is clicked.
- Increment the `m_iCount` variable and update the dialog on each timer event.
- Stop the timer when the `IDC_STOPTIMER` button is clicked.

To implement this additional functionality, perform the following steps:

1. Edit the `OnInitDialog` function, updating the code as in Listing 4.4.

LISTING 4.4. THE UPDATED `OnInitDialog` FUNCTION.

```
 1: BOOL CTimersDlg::OnInitDialog()
 2: {
 3:     CDialog::OnInitDialog();
 4: .
 5: .
 6: .
 7:     // TODO: Add extra initialization here
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Initialize the counter interval
14:     m_iInterval = 100;
15:
16:     // Update the dialog
17:     UpdateData(FALSE);
18:
19:     // Start the clock timer
20:     SetTimer(ID_CLOCK_TIMER, 1000, NULL);
21:
22:     ///////////////////////
23:     // MY CODE ENDS HERE
24:     ///////////////////////
25:
26:     return TRUE;  // return TRUE  unless you set the focus to a
                      ➥control
27: }
```

2. Using the Class Wizard, add a function to the `BN_CLICKED` message on the `IDC_STARTTIME` button. Edit the `OnStarttime` function as in Listing 4.5.

LISTING 4.5. THE `OnStarttime` FUNCTION.

```
 1: void CTimersDlg::OnStarttime()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Update the variables
```

```
10:     UpdateData(TRUE);
11:
12:     // Initialize the count
13:     m_iCount = 0;
14:     // Format the count for displaying
15:     m_sCount.Format("%d", m_iCount);
16:
17:     // Update the dialog
18:     UpdateData(FALSE);
19:     // Start the timer
20:     SetTimer(ID_COUNT_TIMER, m_iInterval, NULL);
21:
22:     ///////////////////////
23:     // MY CODE ENDS HERE
24:     ///////////////////////
25: }
```

3. Using the Class Wizard, add a function to the `BN_CLICKED` message on the `IDC_STOPTIMER` button. Edit the `OnStoptimer` function as in Listing 4.6.

LISTING 4.6. THE `OnStoptimer` FUNCTION.

```
 1: void CTimersDlg::OnStoptimer()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Stop the timer
10:     KillTimer(ID_COUNT_TIMER);
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
```

4. Edit the `OnTimer` function, updating the code as in Listing 4.7.

LISTING 4.7. THE UPDATED `OnTimer` FUNCTION.

```
 1: void CTimersDlg::OnTimer(UINT nIDEvent)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
```

continues

LISTING 4.7. CONTINUED

```
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the current time
10:     CTime curTime = CTime::GetCurrentTime();
11:
12:     // Which timer triggered this event?
13:     switch (nIDEvent)
14:     {
15:         // The clock timer?
16:     case ID_CLOCK_TIMER:
17:         // Display the current time
18:         m_sTime.Format("%d:%d:%d", curTime.GetHour(),
19:             curTime.GetMinute(),
20:             curTime.GetSecond());
21:         break;
22:         // The count timer?
23:     case ID_COUNT_TIMER:
24:         // Increment the count
25:         m_iCount++;
26:         // Format and display the count
27:         m_sCount.Format("%d", m_iCount);
28:         break;
29:     }
30:
31:     // Update the dialog
32:     UpdateData(FALSE);
33:
34:     ///////////////////////
35:     // MY CODE ENDS HERE
36:     ///////////////////////
37:
38:     CDialog::OnTimer(nIDEvent);
39: }
```

In the `OnInitDialog` function, you added the initialization of the `m_iInterval` variable, starting it at 100. This initialization is reflected on the dialog window by calling the `UpdateData` function.

In the `OnStarttime` function, you first synchronize the variables with the control values, allowing you to get the current setting of the `m_iInterval` variable. Next, you initialize the `m_iCount` variable, setting it to `0`, and then format the value in the `m_sCount` `CString` variable, which is updated in the dialog window. The last thing that you do is to start the timer, specifying the `ID_COUNT_TIMER` ID and using the interval from the `m_iInterval` variable.

In the `OnStoptimer` function, all you really need to do is stop the timer. You do this by calling the `KillTimer` function, passing the timer ID as the only argument.

It is in the `OnTimer` function that things begin to get interesting. Here, you still see the code for handling the clock timer event. To add the functionality for the counter timer, you need to determine which timer has triggered this function. The only argument to the `OnTimer` function just happens to be the timer ID. You can use this ID in a switch statement to determine which timer has called this function and to control which set of code is executed. The clock timer code is still the same as it was in Listing 4.2. The counter timer code is placed into its spot in the switch statement, incrementing the counter and then updating the `m_sCount` variable with the new value. You can compile and run your application at this point, and you can specify a timer interval and start the timer running, as in Figure 4.6.

FIGURE 4.6.
A running counter on your application dialog.

Enabling the Stop Button

If you run your application, you'll find that it works well except for one small problem. When you start your second timer, you can't stop it. When you were specifying all the properties of the controls, you disabled the Stop Timer button. Before you can stop the timer, you need to enable this button.

What makes the most sense is enabling the stop button and disabling the start button once the timer starts. Then you reverse the situation when the timer stops again. You can do this in the same way you enabled and disabled controls on Day 2, or you can modify your approach just a little.

Remember that when you added variables to the controls, you added variables to the start and stop buttons. These were not normal variables, but control variables. Instead of getting a pointer to these controls using their IDs, you can work directly with the control variables. Try that now by updating the `OnStarttime` and `OnStoptimer` functions as in Listing 4.8.

LISTING 4.8. THE REVISED `OnStarttime` AND `OnStoptimer` FUNCTIONS.

```
 1: void CTimersDlg::OnStarttime()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Update the variables
10:     UpdateData(TRUE);
11:
12:     // Initialize the count
13:     m_iCount = 0;
14:     // Format the count for displaying
15:     m_sCount.Format("%d", m_iCount);
16:
17:     // Update the dialog
18:     UpdateData(FALSE);
19:     // Start the timer
20:     SetTimer(ID_COUNT_TIMER, m_iInterval, NULL);
21:
22:     // Enable the Stop Timer button
23:     m_cStopTime.EnableWindow(TRUE);
24:     // Disable the Start Timer button
25:     m_cStartTime.EnableWindow(FALSE);
26:
27:     ///////////////////////
28:     // MY CODE ENDS HERE
29:     ///////////////////////
30: }
31:
32: void CTimersDlg::OnStoptimer()
33: {
34:     // TODO: Add your control notification handler code here
35:
36:     ///////////////////////
37:     // MY CODE STARTS HERE
38:     ///////////////////////
39:
40:     // Stop the timer
41:     KillTimer(ID_COUNT_TIMER);
42:
43:     // Disable the Stop Timer button
44:     m_cStopTime.EnableWindow(FALSE);
45:     // Enable the Start Timer button
46:     m_cStartTime.EnableWindow(TRUE);
47:
48:     ///////////////////////
49:     // MY CODE ENDS HERE
```

```
50:     ///////////////////////
51: }
```

Now when you compile and run your application, it looks more like Figure 4.7, where you can start and stop the counter timer. This enables you to play with the timer interval, putting in a variety of time intervals and observing the difference, with the clock ticking above the counter for reference.

FIGURE 4.7.
The finished application.

Summary

Today you learned how to use the timers built into the Windows operating system to trigger your application at various time intervals that you can control. You learned how to use multiple timers in the same application, running them simultaneously and triggering different actions.

In the coming days, you'll learn how to use additional dialog windows to get feedback from the user so that you can integrate that feedback into how your application behaves. After that, you will learn how to a create menus for your applications. Then you will learn how you can work with text and fonts in your applications.

Q&A

Q What is the interval range that I can set for timers in my applications?

A The available range that you can set for timers in your applications is around 55 milliseconds on the short end to 2^{32} - 1 milliseconds, or around 49 1/2 days, on the long end.

Q How many timers can I have running at the same time in my application?

A That depends. There are a limited number of timers available to all applications in the Windows operating system. Although the number that is available should be more than sufficient for all running applications using no more than a handful of timers, if an application goes overboard and begins hogging the timers, the operating system may run out. It could be your application that is denied the use of some timers, or it could be other applications that don't have any to use. As a general

rule, if you use more than two or three timers at the same time, you might want to reconsider your application design and determine if there is another way to design and build your application so that it can work with fewer timers.

Q Is there any way to trigger my application to perform some work when it is idle, instead of using a timer to trigger the work when I think my app might be idle?

A Yes, there is. All Windows applications have an `OnIdle` function that can be used to trigger idle processing. `OnIdle` is discussed later on Day 18, "Doing Multiple Tasks at One Time—Multitasking."

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What did you accomplish by adding the two timer IDs to the resource symbols?
2. What is another way to add these two IDs to the application?
3. How can you tell two timers apart in the `OnTimer` function?
4. How many timer events does your application receive if the timer is set for one second and your application has been busy for one minute, preventing it from receiving any timer event messages?

Exercise

Update your application so that when the counter timer is started, the clock timer is reset to run at the same interval as the counter timer. When the counter timer is stopped, return the clock timer to a one-second interval.

WEEK 1

DAY 5

Getting User Feedback—Adding Dialog Boxes to Your Application

With most applications that you might use, there are numerous situations where the application asks you for information—how you want the application configured or whether you want to save your work before exiting, for example. In most of these situations, the application opens a new window to ask these questions. These windows are called dialog windows.

Dialog windows typically have one or more controls and some text explaining what information the program needs from you. Dialog windows typically do not have a large blank work area, as you find in the main windows of a word processor or a programming editor. All the applications that you have built in the preceding days have been dialog windows, and your projects will continue to be dialog windows for the next few days.

All the dialogs that you have created up to now have been single window dialog applications. Today you are going to learn

- How to use dialog windows in a more flexible way.
- How to call other dialog windows and take the information entered by the user on these windows back to the main application window for use in the application.
- How to use both standard dialogs, such as the message boxes you used in previous days and custom dialogs that you have created.

Using Pre-existing (or System) Dialog Windows

The Windows operating system provides a number of pre-existing dialog windows. Simple dialog windows, also known as message boxes, present the user with a message and provide one to three buttons to click. More complex dialogs, such as the File Open, Save, or Print dialogs, are also provided with Windows. These system (or common) dialogs are created and used with a combination of a variable declaration of a C++ class and a series of interactions with the class instance.

Using Message Boxes

As you learned in the previous days, using message boxes is as simple as making a single function call, passing the message text as the only argument. This results in a message box that displays the message to the user with an icon and gives the user one button to click to acknowledge the message. As you probably know from using other Windows software, you have a whole range of other message box possibilities with various button combinations and various icons that can be displayed.

The `MessageBox` Function

As you have seen in previous days, the `MessageBox` function can be passed one or two arguments. The first argument is the message to be displayed to the user. The second argument, which is completely optional, is displayed in the title bar on the message box. You can use a third argument, which is also optional, to specify the buttons to be presented to the user and the icon to be displayed beside the message. In addition to this third argument, the `MessageBox` function returns a result value that indicates which button was clicked by the user. Through the combination of the third argument and the return value, the `MessageBox` function can provide a whole range of functionality in your Visual C++ applications.

> If you use the third argument to the `MessageBox` function to specify the buttons or the icon to be presented to the user, the second argument (the message box title) is no longer optional. You must provide a value for the title bar of the message box.

The button combinations that you can use in the `MessageBox` function are limited. You do not have the freedom to make up your own button combination. If you get to the point where you need to make up your own, you have to create a custom dialog window that looks like a message box. The button combinations that you can use are listed in Table 5.1.

TABLE 5.1. MESSAGEBOX BUTTON COMBINATION IDS.

ID	*Buttons*
`MB_ABORTRETRYIGNORE`	Abort, Retry, Ignore
`MB_OK`	OK
`MB_OKCANCEL`	OK, Cancel
`MB_RETRYCANCEL`	Retry, Cancel
`MB_YESNO`	Yes, No
`MB_YESNOCANCEL`	Yes, No, Cancel

5

To specify the icon to be displayed, you can add the icon ID to the button combination ID. The icons that are available are listed in Table 5.2. If you want to specify either the icon or the button combination, and you want to use the default for the other, you can just specify the one ID that you want to use.

TABLE 5.2. MESSAGEBOX ICON IDS.

ID	*Icon*
`MB_ICONINFORMATION`	Informational icon
`MB_ICONQUESTION`	Question mark icon
`MB_ICONSTOP`	Stop sign icon
`MB_ICONEXCLAMATION`	Exclamation mark icon

When you do specify a button combination, you want to capture the return value so that you can determine which button the user clicked. The return value is defined as an integer data type; the return value IDs are listed in Table 5.3.

TABLE 5.3. MESSAGEBOX RETURN VALUE IDS.

ID	*Button Clicked*
`IDABORT`	Abort
`IDRETRY`	Retry
`IDIGNORE`	Ignore
`IDYES`	Yes
`IDNO`	No
`IDOK`	OK
`IDCANCEL`	Cancel

Creating a Dialog Application

To get a good understanding of how you can use the `MessageBox` function in your applications to get information from the user, you will build a simple application that uses the `MessageBox` function in a couple of different ways. Your application will have two separate buttons that call two different versions of the `MessageBox` function so that you can see the differences and similarities between the various options of the function. Later in the day, you will add a standard File Open dialog so that you can see how the standard dialogs can be used to allow the user to specify a filename or perform other standard functions. Finally, you will create a custom dialog that allows the user to enter a few different types of values, and you will see how you can read these values from the main application dialog after the user has closed the custom dialog.

To start this application, follow these steps:

1. Create a new MFC AppWizard workspace project, naming it **`Dialogs`**.
2. Choose the same settings as for the previous days' applications, giving the application a title of **`Dialogs`**.
3. Lay out the main application dialog as shown in Figure 5.1 using the properties in Table 5.4.

TABLE 5.4. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Command Button	ID	`IDC_YESNOCANCEL`
	Caption	`&Yes, No, Cancel`
Command Button	ID	`IDC_ABORTRETRYIGNORE`
	Caption	`&Abort, Retry, Ignore`

Object	*Property*	*Setting*
Command Button	ID	`IDC_FILEOPEN`
	Caption	`&File Open`
Command Button	ID	`IDC_BCUSTOMDIALOG`
	Caption	`&Custom Dialog`
Command Button	ID	`IDC_BWHICHOPTION`
	Caption	`&Which Option?`
	Disabled	Checked
Command Button	ID	`IDC_EXIT`
	Caption	`E&xit`
Static Text	ID	`IDC_STATIC`
	Caption	`Dialog Results:`
Edit Box	ID	`IDC_RESULTS`
	Multiline	Checked
	Auto Vscroll	Checked

FIGURE 5.1.

The application main dialog layout.

4. Using the Class Wizard, attach variables to the controls as listed in Table 5.5.

TABLE 5.5. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
IDC_RESULTS	m_sResults	Value	CString
IDC_BWHICHOPTION	m_cWhichOption	Control	CButton

5. Using the Class Wizard, attach code to the Exit button to close the application, as on previous days.

Coding the Message Box Dialogs

For the first command button (the Yes, No, Cancel button), create a function on the clicked event using the Class Wizard, just as you did on previous days. Edit the function on this button, adding the code in Listing 5.1.

LISTING 5.1. THE `OnYesnocancel` FUNCTIONS.

```
 1: void CDialogsDlg::OnYesnocancel()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     int iResults; // This variable will capture the button selection
10:
11:     // Ask the user
12:     iResults = MessageBox("Press the Yes, No, or Cancel button",
13:                 "Yes, No, Cancel Dialog",
14:                 MB_YESNOCANCEL ¦ MB_ICONINFORMATION);
15:
16:     // Determine which button the user clicked
17:     // Give the user a message showing which button was clicked
18:     switch (iResults)
19:     {
20:     case IDYES:    // The Yes button?
21:         m_sResults = "Yes! Yes! Yes!";
22:         break;
23:     case IDNO:    // The No button?
24:         m_sResults = "No, no, no, no, no.";
25:         break;
26:     case IDCANCEL:    // The Cancel button?
27:         m_sResults = "Sorry, canceled.";
28:         break;
29:     }
30:
31:     // Update the dialog
```

```
32:     UpdateData(FALSE);
33:
34:     ///////////////////////
35:     // MY CODE ENDS HERE
36:     ///////////////////////
37: }
```

If you compile and run your application, you can see how selecting the different buttons on the message box can determine the next course of action in your application. If you add a function to the clicked event of the Abort, Retry, Ignore button using the Class Wizard and enter the same code as in Listing 5.1, substituting the `MB_ABORTRETRYIGNORE` and `MB_ICONQUESTION` values and changing the prompts and messages, you can see how this other button combination can be used in the same way.

Both of these control event functions are virtually the same. In each function, there is an integer variable declared to capture the return value from the `MessageBox` function. Next, the `MessageBox` function is called with a message to be displayed to the user, a title for the message box, and a combination of a button combination ID and an icon ID.

When the return value is captured from the `MessageBox` function, that value is passed through a switch statement to determine which value was returned. A message is displayed to the user to indicate which button was clicked on the message box. You can just as easily use one or two `if` statements to control the program execution based on the user's selection, but the return value being an integer lends itself to using a switch statement.

If you compile and run your application at this point, you can click either of the top two buttons and see a message box, as in Figure 5.2. When you click one of the message box buttons, you see a message in the edit box on the main dialog, indicating which button you selected, as in Figure 5.3.

FIGURE 5.2.
The `MessageBox` *with three choices.*

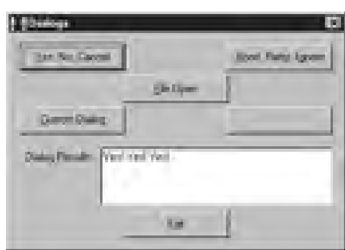


FIGURE 5.3.
A message is displayed based on which button was clicked.

Using Common Dialogs

Using common dialogs is not quite as simple and easy as using the `MessageBox` function, but it's still quite easy. The Microsoft Foundation Classes (MFC) provides several C++ classes for common Windows dialogs. These classes are listed in Table 5.6.

TABLE 5.6. COMMON DIALOG CLASSES.

Class	*Dialog Type*
`CFileDialog`	File selection
`CFontDialog`	Font selection
`CColorDialog`	Color selection
`CPageSetupDialog`	Page setup for printing
`CPrintDialog`	Printing
`CFindReplaceDialog`	Find and Replace

The common dialogs encapsulated in these classes are the standard dialogs that you use every day in most Windows applications to open and save files, configure printing options, print, perform find and replace on documents, and so on. In addition to these choices, a series of OLE common dialog classes provide several common functions to OLE or ActiveX components and applications.

All these dialogs are used in the same manner, although the individual properties and class functions vary according to the dialog functionality. To use one of these dialogs, you must follow these steps:

1. Declare a variable of the class type.
2. Set any properties that need to be configured before displaying the dialog to the user.
3. Call the `DoModal` method of the class to display the dialog to the user.
4. Capture the return value of the `DoModal` method to determine whether the user clicked the OK or Cancel button.
5. If the user clicks the OK button, read any properties that the user may have set when using the dialog.

To better understand how this works, you'll add the `CFileDialog` class to your application. To do this, add a function to the clicked message on the File Open button using the Class Wizard. Edit this function, adding the code in Listing 5.2.

LISTING 5.2. THE `OnFileopen` FUNCTION.

```
 1: void CDialogsDlg::OnFileopen()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     CFileDialog m_ldFile(TRUE);
10:
11:     // Show the File open dialog and capture the result
12:     if (m_ldFile.DoModal() == IDOK)
13:     {
14:         // Get the filename selected
15:         m_sResults = m_ldFile.GetFileName();
16:         // Update the dialog
17:         UpdateData(FALSE);
18:     }
19:
20:     ///////////////////////
21:     // MY CODE ENDS HERE
22:     ///////////////////////
23: }
```

In this code, the first thing that you do is declare an instance of the `CFileDialog` class. This instance is passed `TRUE` as an argument to the class constructor. This tells the class that it is a File Open dialog. If you pass it `FALSE`, it displays as a File Save dialog. There's no real functional difference between these two, only a visual difference. You can pass many more arguments to the constructor, specifying the file extensions to show, the default starting file and location, and filters to use when displaying the files. All the rest of these constructor arguments have default values, so you don't have to supply any of them.

After creating the instance of the File Open dialog, you call its `DoModal` function. This is a member function of the `CDialog` ancestor class, and it is available in all dialog windows. The `DoModal` function displays the File Open dialog to the user, as shown in Figure 5.4. The return value of the `DoModal` function is examined to determine which button the user clicked. If the user clicks the Open button, the `IDOK` value is returned, as with the `MessageBox` function. This is how you can determine whether your application needs to take any action on what the user selected with the dialog window.

FIGURE 5.4.
The File Open dialog.

> There are two modes in which a dialog window can be displayed to the user. The first is as a modal window. A modal window halts all other user interaction while it is displayed. The user cannot do anything else in the application until the dialog is closed. A good example of a modal dialog window is a message box where the user cannot continue working with the application until she clicks one of the buttons on the message box.
>
> The second mode in which a dialog window can be displayed to the user is as a modeless window. A modeless window can be open while the user is doing something else in the application, and it doesn't prevent the user from performing other tasks while the dialog is visible. Good examples of a modeless dialog window are the Find and Find and Replace dialogs in Microsoft Word. These dialog windows can be open and displayed on the screen while you are still editing the document that you are searching.

To display the name of the file selected, you set the `m_sResults` variable to the return value from the `GetFileName` method of the `CFileDialog` class. This method returns only the filename without the directory path or drive name, as shown in Figure 5.5. You can use other class methods for getting the directory path (`GetPathName`) or file extension (`GetFileExt`).

FIGURE 5.5.
Displaying the selected filename.

Creating Your Own Dialog Windows

Now you have an understanding of using standard dialogs. What if you need to create a custom dialog for your application? This task is fairly simple to do because it is mostly a combination of the process that you have already used to create and use the main dialog windows in all your applications and the methods you employed to use the common dialogs. You have to work through a few additional steps, but they are few and you should be comfortable with them soon.

Creating the Dialog Window

For the custom dialog that you will add to your application, you will provide the user with a edit box in which to enter some text and a group of radio buttons from which the user can select one. When the user clicks the OK button, your application will display the text entered by the user in the display area of the main application dialog window. There is another button that the user can, can click to display which one of the radio buttons was selected. This exercise enables you to see how you can use custom dialog windows to gather information from the user and how you can read the user's selections after the dialog window is closed.

To create a custom dialog for your application, you need to

- Add another dialog to your application resources.
- Design the dialog window layout.
- Declare the base class from which the dialog will be inherited.
- Attach variables to the controls on the dialog.

After doing these things, your custom dialog will be ready for your application. To accomplish these tasks, follow these steps:

1. Select the Resource View tab in the project workspace pane.
2. Right-click the Dialogs folder, and select Insert Dialog from the pop-up menu.
3. Right-click the new dialog in the resource tree view, and select Properties from the pop-up menu.
4. Change the object ID for the new dialog to `IDD_MESSAGEDLG`.
5. When editing the new dialog window, do not delete the OK and Cancel buttons. Move them to the location shown in Figure 5.6.

FIGURE 5.6.

The custom dialog window layout.

6. Design the rest of the window using the object properties in Table 5.7.

TABLE 5.7. THE CUSTOM DIALOG CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`Enter a &message:`
Edit Box	ID	`IDC_MESSAGE`
	Multiline	Checked
	Auto Vscroll	Checked
Group Box	ID	`STATIC`
	Caption	`Select an Option`
Radio Button	ID	`IDC_OPTION1`
	Caption	`&Option 1`
	Group	Checked
Radio Button	ID	`IDC_OPTION2`
	Caption	`O&ption 2`
Radio Button	ID	`IDC_OPTION3`
	Caption	`Op&tion 3`
Radio Button	ID	`IDC_OPTION4`
	Caption	`Opt&ion 4`

7. After you design the dialog, open the Class Wizard. You see the dialog in Figure 5.7.

FIGURE 5.7.

The Adding a Class dialog.

8. Leave the selection on this dialog at the default setting of Create a New Class and click OK. Another dialog appears to allow you to specify the name for the new class and the base class from which it is inherited.
9. Enter the class name **CMsgDlg** into the Name field, and make sure that the Base Class is set to CDialog, as shown in Figure 5.8.

FIGURE 5.8.
The New Class dialog.

10. Click OK, leaving the other settings on this dialog at their defaults.
11. Once the Class Wizard opens, attach the variables to the controls on the new dialog as specified in Table 5.8.

TABLE 5.8. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
IDC_MESSAGE	m_sMessage	Value	CString
IDC_OPTION1	m_iOption	Value	int

You should notice two things in the way that you configured the control properties and variables in the custom dialog. First, you should have selected the Group property on only the first of the radio buttons. This designates that all the radio buttons following that one belong to a single group, where only one of the radio buttons may be selected at a time. If you select the Group property on all the radio buttons, they are all independent of each other, allowing you to select all the buttons simultaneously. This property makes them behave somewhat like check boxes, but the primary difference is that the user would find it difficult to uncheck one of these controls due to the default behavior where one radio button in each group is always checked. The other difference is in their appearance; the radio buttons have round selection areas instead of the square areas of check boxes.

The other thing to notice is that you declared a single integer variable for the one radio button with the Group property checked. This variable value is controlled by which radio button is selected. The first radio button causes this variable to have a value of 0, the second sets this variable to 1, and so on. Likewise, if you want to automatically select a particular radio button, you can set this variable to one less than the sequence number of the radio button in the group of radio buttons.

> Because this is the C++ programming language, all numbering begins with 0, not 1. Therefore, the first position in an array or a set of controls is position 0. The second position is position 1. The third position is number 2, and so on.

You have now finished all that you need to do to the second dialog window to make it ready for use. You would expect to need an `UpdateData` or two in the code behind the dialog, but because you didn't remove the OK and Cancel buttons from the dialog, the `UpdateData` call is already performed when the user clicks the OK button. As a result, you don't have to touch any code in this second dialog, only in the first dialog.

Using the Dialog in Your Application

Now that your custom dialog is ready for your application, using it is similar to the way that you use the common dialogs that are built into Windows. First, you have to declare an instance of the custom dialog class, which calls the class constructor and creates an instance of the class. Next, you call the dialog's `DoModal` method and capture the return value of that function. Finally, you read the values of the variables that you associated with the controls on the dialog.

Creating the Dialog Instance

Before you can use your custom dialog in your application, you have to make your main dialog window aware of the custom dialog, its variables, and methods and how your main dialog can interact with your custom dialog. You accomplish this by including the header file for your custom dialog in the main source file for your main application dialog. Follow these steps:

1. Select the File View tab on the workspace pane.
2. Expand the Dialog Files and Source Files folders.
3. Double-click the `DialogsDlg.cpp` file. This opens the source code file for the main application dialog in the editing area of Developer Studio.

4. Scroll to the top of the source code file where the `#include` statements are located, and add an include for the `MsgDlg.h` file before the `DialogsDlg.h` file, as in Listing 5.3.

LISTING 5.3. THE HEADER FILE INCLUDES.

```
 1: // DialogsDlg.cpp : implementation file
 2: //
 3:
 4: #include "stdafx.h"
 5: #include "Dialogs.h"
 6: #include "MsgDlg.h"
 7: #include "DialogsDlg.h"
 8:
 9: #ifdef _DEBUG
10: #define new DEBUG_NEW
11: #undef THIS_FILE
12: static char THIS_FILE[] = __FILE__;
13: #endif
14:
15:///////////////////////////////////////////////////////////////////////
16: // CAboutDlg dialog used for App About
```

It is important that you place the `#include` statement for the `MsgDlg.h` file before the `#include` statement for the `DialogsDlg.h` file. The reason is that you will be adding a variable declaration for your custom dialog to the main dialog class in the main dialog's header file. If the `MsgDlg.h` header file is included after the header file for the main dialog, the compiler will complain loudly and will refuse to compile your application until you move the `#include` of the `MsgDlg.h` file above the `#include` of the `DialogsDlg.h` file.

> The `#include` statement is what is known as a compiler directive in the C and C++ programming languages. What it tells the compiler to do is read the contents of the file named into the source code that is being compiled. It is used to separate class, structure, and function declarations into a file that can be included in any source code that needs to be aware of the information in the header file. For more information on how the `#include` statements work, and why you use them, see Appendix A, "C++ Review."

Now that you have made your main application dialog aware of the custom dialog that you created, you need to declare a variable of your custom dialog. Follow these steps:

1. Select the Class View tab in the workspace pane.
2. Right-click the `CDialogsDlg` class to bring up the pop-up menu.
3. Select Add Member Variable from the pop-up menu.
4. Specify the Variable Type as **`CMsgDlg`**, the Variable Name as **`m_dMsgDlg`**, and the Access as Private. Click OK to add the variable to your main dialog.

If you expand the `CDialogsDlg` class in the tree view, you should see the instance of your custom dialog as a member of the main application dialog class. This means that you are ready to begin using the custom dialog in your application.

Calling the Dialog and Reading the Variables

Now that you have added your custom dialog to the main application dialog as a variable that is always available, not just as a local variable available only within a single function (as with the `CFileDialog` variable), you can add code to use the dialog. To do this, follow these steps:

1. Open the Class Wizard and add a function to the clicked event message of the `IDC_BCUSTOMDIALOG` button.
2. Add a function for the clicked event message (`BN_CLICKED`) for the `IDC_BWHICHOPTION` button.
3. Edit the `OnBcustomdialog` function, adding the code in Listing 5.4.

LISTING 5.4. THE `OnBcustomdialog` FUNCTION.

```
 1: void CDialogsDlg::OnBcustomdialog()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Show the message dialog and capture the result
10:     if (m_dMsgDlg.DoModal () == IDOK)
11:     {
12:         // The user checked OK, display the message the
13:         // user typed in on the message dialog
14:         m_sResults = m_dMsgDlg.m_sMessage;
15:         // Update the dialog
16:         UpdateData(FALSE);
17:         // Enable the Which Option button
```

```
18:         m_cWhichOption.EnableWindow(TRUE);
19:     }
20:
21:     ///////////////////////
22:     // MY CODE ENDS HERE
23:     ///////////////////////
24: }
```

4. Edit the `OnBwhichoption` function, adding the code in Listing 5.5.

LISTING 5.5. THE `OnBwhichoption` FUNCTION.

```
 1: void CDialogsDlg::OnBwhichoption()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Determine which radio button was selected, and display
10:     // a message for the user to show which one was selected.
11:     switch(m_dMsgDlg.m_iOption)
12:     {
13:     case 0:    // Was it the first radio button?
14:         m_sResults = "The first option was selected.";
15:         break;
16:     case 1:    // Was it the second radio button?
17:         m_sResults = "The second option was selected.";
18:         break;
19:     case 2:    // Was it the third radio button?
20:         m_sResults = "The third option was selected.";
21:         break;
22:     case 3:    // Was it the fourth radio button?
23:         m_sResults = "The fourth option was selected.";
24:         break;
25:     default:    // Were none of the radio buttons selected?
26:         m_sResults = "No option was selected.";
27:         break;
28:     }
29:
30:     // Update the dialog
31:     UpdateData(FALSE);
32:
33:     ///////////////////////
34:     // MY CODE ENDS HERE
35:     ///////////////////////
36:
```

In the first listing, you called the `DoModal` method of the custom dialog, which displayed the dialog for the user, waiting for the user to click one of the two buttons on the dialog, as in Figure 5.9. If the user clicks the OK button, you copy the message the user typed in the custom dialog into the edit box variable to be displayed to the user. After updating the dialog display with the new variable values, you enable the Which Option button, as shown in Figure 5.10. If the user clicks the Cancel button, none of this is done. The dialog display is not changed.

FIGURE 5.9.
The custom dialog allows the user to enter a message.

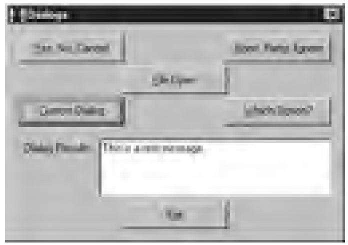


FIGURE 5.10.
The message entered on the custom dialog is displayed for the user.

When the user clicks the Which Option button, you pass the radio button variable on the custom dialog to a switch statement, selecting a message that tells the user which radio button was selected, as shown in Figure 5.11. Notice that in both of these functions, you can access the control variables on the custom dialog directly from the main dialog. That is because the Class Wizard automatically declares the variables associated with controls as public, making them completely accessible outside the dialog class. You can change this by placing a `private:` access specifier where the `public:` access specifier is. You don't want to place anything after the `//{{AFX_DATA` line, where the variables are declared, because the variables are declared within an MFC Class Wizard macro, which enables the Developer Studio wizards to locate and manipulate the variables as needed without interfering with the Visual C++ compiler when you compile your application.

FIGURE 5.11.

The option selected on the custom dialog is displayed for the user.

Summary

Today you learned how you can use additional dialog windows in your application to provide interactive experience for your users. You learned about the options available to you with the simple `MessageBox` function, how you can provide your users a variety of button combinations, and how you can determine which button the user selects. You saw how you can use this information to determine which path to take in your application logic.

You also learned about some of the common dialogs that are built into the Windows operating systems and how they have been encapsulated into C++ classes in the MFC class library. You learned how you can use the File Open dialog to present the user with the standard file selection dialog and how you can determine which file the user selected.

Finally, you learned how you can design your own additional dialogs that you can add to your applications to get information from the user and how you can capture that information and use it in your application.

Q&A

Q There was no code added to the custom dialog. Do I have to design my custom dialogs this way, or can I add code to them?

A The custom dialog windows are no different from the main dialog windows that you have been using in all your applications so far. If you need to control the behavior of the dialog on an interactive basis, you can put as much code into the dialog as you need. You didn't add any code to the custom dialog today because there wasn't any need to add any code. The only functionality that the dialog needed to perform was calling the `UpdateData` function before closing, which is automatically done by the `OnOK` function. Because you did not delete the OK and Cancel buttons, you already had this functionality built in.

Q What happens if I specify two or more button combinations in the same `MessageBox` function call?

A Nothing happens. Your application compiles just fine, but when the `MessageBox` function is called, nothing happens. The message box does not open, and the user does not get to answer the question you are presenting.

Q How can I integrate the File Open dialog into my application where it opens in a specific directory that I specify?

A The `CFileDialog` class has a public property called `m_ofn`. This property is a structure that contains numerous attributes of the File Open dialog, including the initial directory. This structure is defined as the `OPENFILENAME` structure in Listing 5.6.

LISTING 5.6. THE `OPENFILENAME` STRUCTURE.

```
 1: typedef struct tagOFN { // ofn
 2:     DWORD         lStructSize;
 3:     HWND          hwndOwner;
 4:     HINSTANCE     hInstance;
 5:     LPCTSTR       lpstrFilter;
 6:     LPTSTR        lpstrCustomFilter;
 7:     DWORD         nMaxCustFilter;
 8:     DWORD         nFilterIndex;
 9:     LPTSTR        lpstrFile;
10:     DWORD         nMaxFile;
11:     LPTSTR        lpstrFileTitle;
12:     DWORD         nMaxFileTitle;
13:     LPCTSTR       lpstrInitialDir;
14:     LPCTSTR       lpstrTitle;
15:     DWORD         Flags;
16:     WORD          nFileOffset;
17:     WORD          nFileExtension;
18:     LPCTSTR       lpstrDefExt;
19:     DWORD         lCustData;
20:     LPOFNHOOKPROC lpfnHook;
21:     LPCTSTR       lpTemplateName;
22: } OPENFILENAME;
```

You can set any of these attributes before calling the `DoModal` class method to control the behavior of the File Open dialog. For instance, if you set the starting directory to `C:\Temp` before calling the `DoModal` method, as in Listing 5.7, the File Open dialog opens in that directory.

LISTING 5.7. THE REVISED `OnFileopen` FUNCTION.

```
 1: void CDialogsDlg::OnFileopen()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     CFileDialog m_ldFile(TRUE);
10:
11:     // Initialize the starting directory
12:     m_ldFile.m_ofn.lpstrInitialDir = "C:\\Temp\\";
13:
14:     // Show the File open dialog and capture the result
15:     if (m_ldFile.DoModal() == IDOK)
16:     {
17:         // Get the filename selected
18:         m_sResults = m_ldFile.GetFileName();
19:         // Update the dialog
20:         UpdateData(FALSE);
21:     }
22:
23:     ///////////////////////
24:     // MY CODE ENDS HERE
25:     ///////////////////////
26: }
```


Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What are the possible return codes that your application might receive from the `MessageBox` function call when you specify the `MB_RETRYCANCEL` button combination?
2. What are the common dialogs that are built into the Windows operating systems that are defined as MFC classes?

3. What is the difference between a modal dialog and a modeless dialog?
4. How can you display a File Save dialog for the user instead of the File Open dialog that you did have in your application?
5. Why did you not need to create any functions and add any code to your custom dialog?

Exercises

1. Modify your application so that it includes the directory with the filename in the application. (Hint: The `GetFileName` function returns the path and filename that was selected in the File Open dialog.)
2. Add a button on the custom dialog that calls the `MessageBox` function with a Yes or No selection. Pass the result back to the main application dialog.

DAY 6

Creating Menus for Your Application

Most Windows applications use pull-down menus to provide the user a number of functions without having to provide buttons on the window. This enables you to provide your users a large amount of functionality while preserving most of your valuable screen real estate for other stuff.

Today you will learn

- How to create menus for your Visual C++ application
- How to attach a menu to your application's main dialog window
- How to call application functions from a menu
- How to create a pop-up menu that can be triggered with the right mouse button
- How to set up accelerator keys for keyboard shortcuts to menus

Menus

Back when the first computer terminals were introduced and users began using computer software, even on large mainframe systems software developers found the need to provide the users with some sort of menu of the functions that the computer could perform. These early menus were crude by today's standards and were difficult to use and navigate. Menus have progressed since then; they've become standardized in how they are used and easy to learn.

The software designers that first came up with the idea of a graphical user interface (GUI) planned to make computer systems and applications easier to learn by making everything behave in a consistent manner. Menus used for selecting application functionality were one part of the GUI design that could be more easily learned if they all worked the same. As a result, a number of standard menu styles were developed.

Menu Styles

The first menu styles that were standardized are the pull-down and cascading menus. These are the menus with the categories all listed in a row across the top of the application window. If you select one of the categories, a menu drops down below the category, with a number of menu entries that can be selected to trigger various functions in the application.

A variation on this menu style is the *cascading menu*, which has another submenu that opens to the right of a menu entry. This submenu is similar to the pull-down menu, with a number of entries that trigger application functions. The menu designers placed no limit on how many cascading menus can be strung together, but it quickly became clear to most developers that more than two cascading levels is a little unwieldy.

Eventually, a third style of menu was developed, called a *pop-up* or *context menu*—a menu that pops up in the middle of the application area, floating freely above the application work area. This is also called a context menu because the specific menu that pops up is dependent on the selected object or workspace area where the cursor or mouse pointer is.

Keyboard Shortcut–Enabling Menus

When users began working with keyboard-intensive applications, such as word processors, it was discovered that taking your hands off the keyboard to use the mouse to make menu selections dramatically reduced productivity. Software designers decided that they needed to add keyboard shortcuts for the various menu entries (especially the most frequently used menu options). For this reason, keyboard shortcuts (accelerators) and hotkeys were added.

Hotkeys are letters that are underlined in each menu entry. If you press the Alt key with the underlined letter, you can select the menu entry that contains the underlined letter. This is a means of navigating application menus without taking your hands off the keyboard.

For more advanced users, application designers added *keyboard shortcuts*, or *accelerators*. An accelerator is a single key combination that you can press to trigger an application function instead of having to navigate through the application menus. This allows advanced users to avoid the overhead of using menus for the most common application functions. To enable users to learn what accelerators are available in an application, the key combination is placed on the menu entry that it can be used to replace, positioned at the right edge of the menu window.

Menu Standards and Conventions

Although there are no standards in how menus are designed, there are a number of conventions for how they are designed and organized. All these conventions are available in *Windows Interface Guidelines for Software Design*, published by Microsoft for use by Windows software developers. The purpose of this publication is to facilitate the development of consistent application behaviors, which will help accomplish one of the primary goals behind the development of GUI systems. The conventions are as follows:

- Use single-word menu categories across the top menu bar. A two-word category can easily be mistaken for two one-word categories.
- The File menu is located as the first menu on the left. It contains all file-oriented functions (such as New, Open, Save, Print, and so on), as well as the Exit function. The Exit option is located at the bottom of the menu, separated from the rest of the menu entries by a border.
- The Edit menu is next to the File menu. The Edit menu contains all editing functions such as Copy, Cut, Paste, Undo, Redo, and so on.
- The View menu contains menu entries that control and affect the appearance of the application work area.
- The Window menu is used in Multiple Document Interface (MDI) style applications. This has functions for controlling the child windows, selecting the current window, and altering the layout. This menu is the next-to-last menu from the right end of the menu bar.
- The Help menu is the final menu on the right end of the menu bar. It contains menu entries that provide instruction or documentation on the application. If the application has any copyrighted or corporate information that needs to be available for viewing, this should be located as the final entry on this menu, labeled About *<application name>*.

Designing Menus

Menus are defined as a resource in Visual C++ applications. Because they are a resource, you can design menus in the Visual C++ editor through the Resource View tab on the workspace pane. When you first create a dialog-style application, there won't be a menu folder in the resource tree, but you can change that.

> Various aspects of Windows applications are considered to be resources, including window layouts, menus, toolbars, images, text strings, accelerators, and so on. All these features are organized in what is known as a *resource file*, which is used by the Visual C++ compiler to create these objects from their definitions. The resource file is a text file with an `.rc` filename extension and contains a textual description of all the various objects, including IDs, captions, dimensions, and so on.
>
> Some resources, such as images and sounds, cannot be described in text, but have to be stored in a binary format. These resources are stored in individual files, with the filenames and locations included in the resource file.

Creating a Menu

Creating a menu is not difficult. You will follow several steps:

1. Create the application that will house the menu.
2. Add a menu resource to your project.
3. Customize the menu resource to include the menu items for your application.
4. Add functionality to your menu by connecting routines to your menu items.

Creating the Application

For the example in this chapter, you will create a simple dialog-style application that contains a single button and a menu. To create your application, do the following:

1. Create a new MFC AppWizard application, naming the project **`Menus`**.
2. Select the default AppWizard settings on all screens. For the dialog title, enter **`Menus`**.
3. When the AppWizard has generated your application shell, delete all the controls from the dialog.
4. Add a single button to the dialog. Name the button `IDC_EXIT`, and specify the caption as `E&xit`.

5. Add a function to the button using the Class Wizard. Change the code in this function to call `OnOK`. Remember, the `OnOK` function causes the application to close.

If you don't remember how to add the `OnOK` function, review the section "Closing the Application" on Day 2, "Using Controls in Your Application," for an example.

Adding and Customizing a Menu

Now that you have the basic application built, it's time to start creating a menu for the application. To create a menu, you will first add a menu resource to your project. When you add the resource, Visual C++ automatically invokes the Menu Designer, which allows you to customize the menu. The following steps show you how to add and customize a menu:

1. Select the Resource View tab in the workspace pane.
2. Select the project resources folder at the top of the tree; in your example, this is Menus.
3. Right-click the mouse to bring up a pop-up menu.
4. Select Insert from the pop-up menu.
5. In the Insert Resource dialog that opens, select Menu from the list of available resources, as in Figure 6.1. Click the New button.

FIGURE 6.1.
The Insert Resource dialog.

6. The Menu Designer opens in the editing area of Developer Studio. The first menu spot is highlighted, as shown in Figure 6.2.

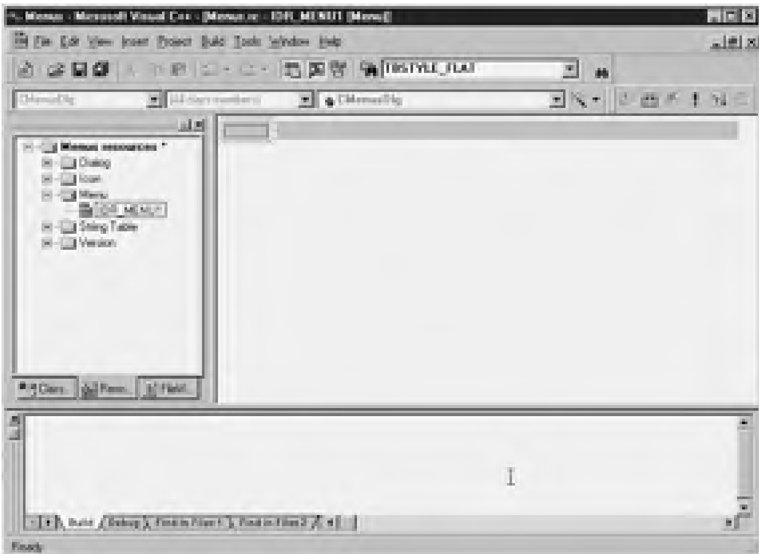


FIGURE 6.2.
An empty menu.

At this point, you have created the menu resource and you are ready to customize it by adding menu items. To add a menu item, follow these steps:

1. Right-click the mouse on the highlighted area and select Properties from the pop-up menu.
2. Enter the menu item's Caption. For this example, enter **`&File`** and close the Properties dialog.

> You are in the menu Properties dialog to specify the text that the user will see on the menu bar while the application is running. Because the Pop-up check box is checked (by default on any menu items on the top-level menu bar), this menu element doesn't trigger any application functionality and thus doesn't need to have an object ID assigned to it.

3. The first drop-down menu location is highlighted. To add this menu item, right-click the mouse again on the highlighted area and select Properties from the pop-up menu.
4. Enter an ID and caption for the menu item. For this example, enter **`IDM_FILE_HELLO`** for the ID and **`&Hello`** for the Caption. Close the dialog.

This time in the menu Properties dialog, you not only specify the text that the user will see when the menu is opened from the menu bar, but you also specify the object ID that will be used in the event message handler to determine what function receives each of the menu events.

At this point you have created a menu with a single menu item. You can continue to add menu items by repeating steps 3 and 4 of the preceding list for each of the highlighted areas. You can also add separators onto the menu. A *separator* is a dividing line that runs across the menu to separate two functional areas of menu selections. To add a separator, perform the following steps:

FIGURE 6.3.

Specifying a menu separator.

1. Select the highlighted area where you want the separator to be placed. In the example you created, the second drop-down menu location should be highlighted. Open the properties dialog as you did in step 3 in the preceding list. To add a separator, simply select the Separator option, as shown in Figure 6.3, and close the dialog.

To complete your sample program, follow the same steps I just described to add an Exit item to your File menu and a second menu called Help with one menu item called About. The following steps, which resemble the preceding list of steps, walk you through adding these additional items:

1. Open the properties dialog for the third drop-down location and specify the ID as `IDM_FILE_EXIT` and the caption as `E&xit`. Close the dialog.
2. Select the second top-level menu location and open the properties dialog. Specify the caption as `&Help` and close the dialog.
3. Open the properties dialog for the first drop-down location on the second top-level menu. Specify the ID as `ID_HELP_ABOUT` and the caption as `&About`. Close the dialog.

At this point, your menu is created; however, it is not attached to your application.

Attaching the Menu to Your Dialog Window

You now have a menu that you can use in your application. If you compile and run your application at this point, however, the menu doesn't appear. You still need to attach the menu to your dialog window. You can attach a menu by following these steps:

1. Open the dialog painter by double-clicking the primary application dialog in the Dialog folder in the Workspace pane. For this example, double-click on `IDD_MENUS_DIALOG`.
2. Select the entire dialog window, making sure that no controls are selected, and open the dialog's properties dialog. (What you are doing is opening the properties for the dialog window itself, not for any of the controls that might be on the window.)
3. Select the menu you have designed from the Menu drop-down list box, as shown in Figure 6.4.

FIGURE 6.4.
Attaching the menu to the dialog window.

If you compile and run your application, you find that the menu is attached to the application dialog, as shown in Figure 6.5. You can select menu entries as you do with any other Windows application—with one small difference. At this point, when you select one of the menu entries, nothing happens. You still need to attach functionality to your menu.

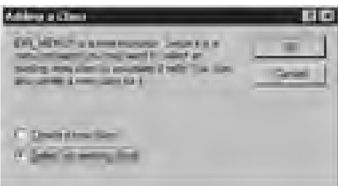
FIGURE 6.5.
The menu is now part of the application dialog.

Attaching Functionality to Menu Entries

Now that you have a menu as part of your application, it sure would be nice if it actually did something. Well, before your menu can do anything, you have to tell it what to do, just like everything else in your Visual C++ applications. To attach some functionality to your menu, follow these steps:

FIGURE 6.6.

The menu is now part of the application.



1. Open the Menu Designer to your menu.
2. Open the Class Wizard from the View menu.
3. The Adding a Class dialog is displayed for you, just as it was yesterday when you added a second dialog. Leave the dialog selection on Select an Existing Class and click OK (see Figure 6.6).

   Yesterday, when you were adding a second dialog window to your application, you needed to create a new C++ class for that window. For today's menu, you want to attach it to the existing C++ class for the dialog window to which the menu is attached.
4. Choose the C++ class of the primary dialog window from the list of available classes in the Select Class dialog. For this example, select `CMenusDlg`, as shown in Figure 6.7. This tells Visual C++ that all the functionality that you will call from the various menu entries is part of the same dialog class of the window that it's attached to.

FIGURE 6.7.

The Select Class dialog.

For the menu elements that you want to use to trigger new functions in your application, you can add event-handler functions through the Class Wizard, just as you can with controls that you place on the dialog window.

For this example, add a function for the `IDM_FILE_HELLO` object (the Hello menu) on the `COMMAND` event message. Name the function `OnHello` and add the code in Listing 6.1 to the function.

LISTING 6.1. THE ONHELLO FUNCTION.

```
 1: void CMenusDlg::OnHello()
 2: {
 3:     // TODO: Add your command handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Display a message for the user
10:     MessageBox("Hello there", "Hello");
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
```

The `COMMAND` event message is the message that is passed to the application window when a menu entry is selected. Placing a function on this event message has the same effect as placing a function on the menu entry selection.

You can call existing event handlers from menu elements by adding the existing function to the menu `COMMAND` event. You can do this by adding a function to the menu object ID and then specifying the existing function name instead of accepting the suggested function name.

To reuse the `OnExit` function for the Exit menu element, reopen the Menu Designer and then reopen the Class Wizard. When the Class Wizard is displayed, add a function for the `IDM_FILE_EXIT` object on the `COMMAND` event message. Do not accept the default function name presented to you by the Class Wizard. Enter the function name **`OnExit`**. This automatically attaches the existing `OnExit` function that you created with your Exit button earlier.

To round out your example's functionality, add a function to the `ID_HELP_ABOUT` object on the `COMMAND` event message. Edit the function as in Listing 6.2.

LISTING 6.2. THE *ONHELPABOUT* FUNCTION.

```
1: void CMenusDlg::OnHelpAbout()
2: {
3:     // TODO: Add your command handler code here
4:
```

```
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Declare an instance of the About window
10:     CAboutDlg dlgAbout;
11:
12:     // Show the About window
13:     dlgAbout.DoModal();
14:
15:     ///////////////////////
16:     // MY CODE ENDS HERE
17:     ///////////////////////
18: }
```

You attached the File|Exit menu entry to an existing function that closes the application. On the File|Hello, you added a new function that called the `MessageBox` function to display a simple message to the user. With Help|About, you added another function that declared an instance of the About dialog window and called its `DoModal` method.

If you compile and run your application, you find that all the menu entries are working. If you select Help|About, as shown in Figure 6.8, you see the application About dialog (see Figure 6.9). If you select File|Hello, you see a `Hello there` message box, as shown in Figure 6.10. And if you select File|Exit, your application closes.

FIGURE 6.8.

The Help|About menu entry.

FIGURE 6.9.

The About dialog.

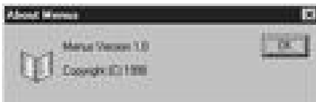


FIGURE 6.10.

The Hello there message box.

Creating Pop-Up Menus

Most Windows applications have what are called either pop-up or context menus, which are triggered by the user right-clicking an object. These are called *pop-up menus* because they pop up in the middle of the application area, not attached to a menu bar, the window frame, or anything else on the computer screen (not counting the mouse pointer). These menus are often referred to as *context menus* because the contents of a menu depend on the context in which it is opened; the elements available on the menu depend on what objects are currently selected in the application or what the mouse pointer is positioned over.

To provide a pop-up menu in your application, you have two approaches available. You can either design a menu specifically for use as a pop-up menu, or you can use one of the pull-down menus from the primary menu that you have already designed. If you design a menu specifically for use as a pop-up menu, you will need to skip the top-level, menu bar element by placing a space or some other text in the caption, knowing that it will not be seen. You will see how this works when you build a custom menu specifically for use as a pop-up menu on Day 11, “Creating Multiple Document Interface Applications,” in the section “Adding a Context Menu.”

Every drop-down portion of a menu can also be used as a pop-up menu. To use it in this way, you must get a handle to the submenu (the drop-down menu) and then call the `TrackPopupMenu` function on the submenu. The rest of the pop-up menu functionality is already covered in the other menu building and coding that you have already done. To add a pop-up menu to your application, follow these steps:

1. Using the Class Wizard, add a function for the `WM_CONTEXTMENU` event message in your dialog window.

There are two dialog event messages that you can use to trigger your context menu. The event that you’d expect to use is the `WM_RBUTTONDOWN` event, which is triggered by the user right-clicking. The other event that can (and should) be used is the `WM_CONTEXTMENU` event, which is intended for use specifically to trigger a context menu. This event is triggered by a couple user actions: One of these is the release of the right mouse button, and another is the pressing of the context menu button on one of the newer Windows-enabled keyboards.

2. Edit the function, adding the code in Listing 6.3.

LISTING 6.3. THE *ONCONTEXTMENU* FUNCTION.

```
 1: void CMenusDlg:: OnContextMenu(CWnd* pWnd, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Declare local variables
10:     CMenu *m_lMenu;     // A pointer to the menu
11:     CPoint m_pPoint;    // A copy of the mouse position
12:
13:     // Copy the mouse position to a local variable
14:     m_pPoint = point;
15:     // Convert the position to a screen position
16:     ClientToScreen(&m_pPoint);
17:     // Get a pointer to the window menu
18:     m_lMenu - GetMenu();
19:     // Get a pointer to the first submenu
20:     m_lMenu = m_lMenu->GetSubMenu(0);
21:     // Show the Popup Menu
22:     m_lMenu->TrackPopupMenu(TPM_CENTERALIGN + TPM_LEFTBUTTON,
23:         m_pPoint.x, m_pPoint.y, this, NULL);
24:
25:     ///////////////////////
26:     // MY CODE ENDS HERE
27:     ///////////////////////
28: }
```

In Listing 6.3, the first thing that you did was make a copy of the mouse position. This mouse position is a relative position within the window area. It must be converted to an absolute position on the entire screen area for displaying the pop-up menu. If you don't convert the position coordinates, you can't predict where your pop-up menu will appear.

After you convert the position to an absolute position, you get a pointer to the window menu. This pointer should always be a local pointer within the function where you are going to use it because the location of the menu might change as the application runs. From the menu pointer, you next get a pointer to the first drop-down menu (submenu numbering begins with 0, like just about everything else in C/C++). After you have a pointer to the submenu, you can treat it as a regular `CMenu` class instance.

The final piece in this puzzle is the call to the `CMenu` member function `TrackPopupMenu`. This function takes five arguments and uses them to determine where and how to show

the pop-up menu. The first argument is a combination of two flags. The first flag, TPM_CENTERALIGN, centers the pop-up menu on the mouse point. You can also use TPM_LEFTALIGN or TPM_RIGHTALIGN instead. These flags line up the left or right edge of the pop-up menu with the mouse position. The second part of this flag combination is TPM_LEFTBUTTON, which makes the pop-up menu trigger from the left mouse button. You can also use TPM_RIGHTBUTTON to make the menu trigger from the right mouse button.

The second and third arguments to the TrackPopupMenu function specify the screen position for the pop-up menu. This is the absolute position on the screen, not a relative position within the window area. The fourth argument is a pointer to the window that receives the menu command messages. The final argument is a rectangle that the user can click without closing the pop-up menu. By passing NULL, you specify that if the user clicks outside the pop-up menu, the menu closes. This code enables you to include a pop-up menu in your application, as shown in Figure 6.11.

FIGURE 6.11.
The pop-up menu in action.

Creating a Menu with Accelerators

One of the original keyboard shortcuts for selecting menu entries were accelerator keys. As mentioned earlier in the chapter, accelerator keys are specific key combinations, usually the Ctrl key combined with another key, or function keys, that are unique within the entire application. Each of these key combinations triggers one menu event function.

The way that accelerator keys work is similar to the way menus work. They are also an application resource that is defined in a table in the resource tab of the workspace pane. Each table entry has an object ID and a key code combination. After you define the accelerators, you can attach functionality to the object IDs. You can also assign accelerator entries the same object ID as the corresponding menu entry so that you have to define only a single entry in the application message map.

After you define all your accelerator keys, you can add the key combination to the menu entry so that the user will know about the accelerator key combination. Add \t to the end of the menu entry caption, followed by the key combination. The \t is replaced in the menu display by a tab, which separates the menu caption from the accelerator key combination.

Unfortunately, accelerator keys don't work in dialog-style windows, so you cannot add them to today's application. You will learn how to attach accelerator keys to menus in a few days when you learn about single and multi-document interface style applications.

Summary

Today you learned about menus in Visual C++ applications. You learned how to use the tools in Visual C++ to create a menu for use in your application and then how to attach the menu to a window in your application. After you had the menu attached to your window, you learned how to attach functionality to the various menu entries. Later in the day, you learned how you can use a portion of your menu as a pop-up, or context, menu. Finally, you learned how accelerator keys are added to most applications.

Q&A

Q Do I have to name my menu items the same names everyone else uses? For example, a lot of applications use File and Help. Can I name my menus something else?

A You can name your top-level menus anything you want. However, there are accepted menu name conventions that place all file-oriented functionality under a menu labeled File and all help-related functionality under a menu labeled Help. If you have a menu with entries such as Broccoli, Corn, and Carrots, you will probably want to call the menu Vegetables, although an equally valid label would be Food or Plants. In general, if you want to make your application easy for your users to learn, you will want to use menu labels that make sense for the entries o n the pull-down portion of the menu.

Q Why can't I specify a single character as an accelerator key?

A The single character would trigger the `WM_KEY` messages, not the menu messages. When the designers of Windows were deciding how accelerator keys would work, they decided that single-character keys would most likely be input to the active application. If they had allowed single-character accelerators, Windows wouldn't be able to determine whether the character was input or a shortcut. By requiring a key combination (with the exception of function keys), the designers ensured that Windows won't have to make this determination.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions are provided in Appendix B, "Answers."

Quiz

1. What event message does a menu selection send to the window message queue?
2. How do you attach a menu to a dialog window?
3. Which existing class do you specify for handling event messages for the menu?
4. What event message should a pop-up menu be triggered by?

Exercises

1. Add a button to the main window and have it call the same function as the Hello menu entry.
2. Add a pop-up menu to your application that uses the Help drop-down menu as the pop-up menu.

WEEK 1

DAY 7

Working with Text and Fonts

In most Windows applications, you don't need to worry about specifying fonts, much less their weight, height, and so on. If you don't specify the font to be used, Windows supplies a default font for your application. If you do need to use a particular font, you can specify a font to use for a particular dialog window through the dialog properties. Sometimes, however, you want or need to control the font used in your application. You might need to change the font being used or allow the user to select a font to use in a particular instance. It is for those circumstances that you will learn how to change and list fonts today. Among the things that you will learn are

- How to build a list of available fonts.
- How to specify a font for use.
- How to change fonts dynamically.

Finding and Using Fonts

One of the first things that you need to know when working with fonts is that not every system that your applications run on will have the same fonts installed. Fonts are specified in files that can be installed and removed from Windows systems with relative ease. Every computer user can customize his system with whatever combination of fonts he wants. If you specify a font that doesn't exist on the system, Windows will choose either the system default font or what the operating system considers to be a reasonably close alternative font.

What you can do instead is ask the operating system what fonts are available. This method allows you to make your own decisions on which font to use or let the user make the decision. When you ask what fonts are available, you can limit the types of fonts that are listed, or you can choose to list them all and select various fonts based on various attributes.

Listing the Available Fonts

To get a list of all available fonts on a computer, you call a Windows API (Application Programming Interface) function called `EnumFontFamiliesEx`. This function tells Windows that you want a list of the fonts on the system. Before you start using this function and expecting it to pass you a big list of available fonts, you need to understand how it gives you the list.

Callback Functions

One of the key arguments to the `EnumFontFamiliesEx` function is the address of another function. This second function is what is known as a callback function, which is called by the operating system. For almost every enumeration function in the Windows operating system, you pass the address of a callback function as an argument because the callback function is called once for each of the elements in the enumerated list. In other words, you have to include a function in your application to receive each individual font that is on the system and then build the list of fonts yourself.

When you create this function to receive each font and build your list, you cannot define your callback function in any way you want. All callback functions are already defined in the Windows API. You have to use a specific type of callback function to receive the list of fonts. For getting a list of fonts, the function type is `EnumFontFamProc`. This function type specifies how your function must be defined, what its arguments must be, and what type of return value it must return. It does not specify what your function should be named or how it needs to work internally. These aspects are left completely up to you.

The `EnumFontFamiliesEx` Function

The `EnumFontFamiliesEx` function, which you call to request the list of available fonts, takes five arguments. A typical use of this function follows:

```
// Create a device context variable
CClientDC dc (this);
// Declare a LOGFONT structure
LOGFONT lLogFont;

// Specify the character set
lLogFont.lfCharSet = DEFAULT_CHARSET;
// Specify all fonts
lLogFont.lfFaceName[0] = NULL;
// Must be zero unless Hebrew or Arabic
lLogFont.lfPitchAndFamily = 0;
// Enumerate the font families
::EnumFontFamiliesEx((HDC) dc, &lLogFont,
(FONTENUMPROC) EnumFontFamProc, (LPARAM) this, 0);
```

The first argument is a device context, which can be an instance of the `CClientDC` class. Every application running within the Windows operating system has a device context. The device context provides a lot of necessary information to the operating system about what is available to the application and what is not.

The second argument is a pointer to a `LOGFONT` structure. This structure contains information about the fonts that you want listed. You can specify in this structure which character set you want to list or whether you want all the fonts in a particular font family. If you want all the fonts on the system, you pass `NULL` in the place of this argument.

The third argument is the address of the callback function that will be used to build your list of fonts. Passing the address of your callback function is a simple matter of using the function name as the argument. The Visual C++ compiler takes care of replacing the function name with the function address. However, you do need to cast the function as the type of callback function that the function requires.

The fourth argument is a `LPARAM` value that will be passed to the callback function. This parameter is not used by Windows but provides your callback function with a context in which to build the font list. In the example, the value being passed is a pointer to the window in which the code is being run. This way, the callback function can use this pointer to access any structures it needs to build the list of fonts. This pointer can also be the first node in a linked list of fonts or other such structure.

The fifth and final argument is always `0`. This reserved argument may be used in future versions of Windows, but for now, it must be `0` so that your application passes a value that won't cause the function to misbehave.

The `EnumFontFamProc` Function Type

When you create your callback function, it must be defined as an independent function, not as a member of any C++ class. A typical EnumFontFamProc function declaration follows:

```
int CALLBACK EnumFontFamProc(
LPENUMLOGFONT lpelf,
LPNEWTEXTMETRIC lpntm,
DWORD nFontType,
long lParam)
{
    // Create a pointer to the dialog window
    CMyDlg* pWnd = (CMyDlg*) lParam;

    // Add the font name to the list box
    pWnd->m_ctlFontList.AddString(lpelf->elfLogFont.lfFaceName);
    // Return 1 to continue font enumeration
    return 1;
}
```

The first argument to this function is a pointer to an ENUMLOGFONTEX structure. This structure contains information about the logical attributes of the font, including the font name, style, and script. You may have numerous fonts listed with the same name but different styles. You can have one for normal, one for bold, one for italic, and one for bold italic.

The second argument is a pointer to a NEWTEXTMETRICEX structure. This structure contains information about the physical attributes of the font, such as height, width, and space around the font. These values are all relative in nature because they need to scale as the font is made larger or smaller.

The third argument is a flag that specifies the type of font. This value may contain a combination of the following values:

- DEVICE_FONTYPE
- RASTER_FONTYPE
- TRUETYPE_FONTYPE

Finally, the fourth argument is the value that was passed into the EnumFontFamiliesEx function. In the example, it was a pointer to the dialog on which the list of fonts is being built. If you cast this value as a pointer to the dialog, the function can access a list box control to add the font names.

The return value from this function determines whether the listing of fonts continues. If 0 is returned from this function, the operating system quits listing the available fonts. If 1 is returned, the operating system continues to list the available fonts.

Using a Font

To use a particular font in an application, you call an instance of the `CFont` class. By calling the `CreateFont` method, you can specify the font to be used, along with the size, style, and orientation. Once you've created a font, you can tell a control or window to use the font by calling the object's `SetFont` method. An example of this process follows:

```
CFont m_fFont;    // The font to be used

// Create the font to be used
m_fFont.CreateFont(12, 0, 0, 0, FW_NORMAL,
        0, 0, 0, DEFAULT_CHARSET, OUT_CHARACTER_PRECIS,
        CLIP_CHARACTER_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH ¦
        FF_DONTCARE, m_sFontName);

// Set the font for the display area
m_ctlDisplayText.SetFont(&m_fFont);
```

> **Tip:** The `CFont` variable used in the previous code should be declared as a member variable of the class in which this code is placed. In the sample code, it is declared above where it is used to show how it is declared. This variable should not be declared or used as a local variable in a function.

Seems simple enough—just two function calls—but that `CreateFont` function needs an awful lot of arguments passed to it. It is these arguments that make the `CreateFont` method a flexible function with a large amount of functionality. Once you create the font, using it is a simple matter of passing the font to the `SetFont` method, which is a member of the `CWnd` class and thus available to all window and control classes in Visual C++. This means that you can use this technique on any visible object within a Visual C++ application.

To understand how the `CreateFont` function works, let's look at the individual arguments that you have to pass to it. The function is defined as

```
BOOL CreateFont(
int nHeight,
    int nWidth,
    int nEscapement,
    int nOrientation,
    int nWeight,
    BYTE bItalic,
    BYTE bUnderline,
    BYTE cStrikeOut,
    BYTE nCharSet,
```

```
    BYTE nOutPrecision,
    BYTE nClipPrecision,
    BYTE nQuality,
    BYTE nPitchAndFamily,
    LPCTSTR lpszFaceName);
```

The first of these arguments, `nHeight`, specifies the height of the font to be used. This logical value is translated into a physical value. If the value is `0`, a reasonable default value is used. If the value is greater or less than `0`, the absolute height is converted into device units. It is key to understand that height values of `10` and `-10` are basically the same.

The second argument, `nWidth`, specifies the average width of the characters in the font. This logical value is translated into a physical value in much the same way as the height is.

The third argument, `nEscapement`, determines the angle at which the text will be printed. This value is specified in 0.1-degree units in a counterclockwise pattern. If you want to print vertical text that reads from bottom to top, you supply `900` as the value for this argument. For printing normal horizontal text that flows from left to right, supply `0` as this value.

The fourth argument, `nOrientation`, determines the angle of each individual character in the font. This works on the same basis as the previous argument, but it controls the output on a character basis, not a line-of-text basis. To print upside-down characters, set this value to `1800`. To print characters on their backs, set this value to `900`.

The fifth argument, `nWeight`, specifies the weight, or boldness, of the font. This can be any value from `0` to `1000`, with `1000` being heavily bolded. You can use constants defined for this argument to control this value with ease and consistency. These constants are listed in Table 7.1.

TABLE 7.1. FONT WEIGHT CONSTANTS.

Constant	*Value*
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500

Constant	Value
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

The actual interpretation and availability of these weights depend on the font. Some fonts only have FW_NORMAL, FW_REGULAR, and FW_BOLD weights. If you specify FW_DONTCARE, a default weight is used, just as with most of the rest of the arguments.

The sixth argument, bItalic, specifies whether the font is to be italicized. This is a boolean value; 0 indicates that the font is not italicized, and any other value indicates that the font is italicized.

The seventh argument, bUnderline, specifies whether the font is to be underlined. This is also a boolean value; 0 indicates that the font is not underlined, and any other value indicates that the font is underlined.

The eighth argument, cStrikeOut, specifies whether the characters in the font are displayed with a line through the character. This is another boolean value using a non-zero value as TRUE and 0 as FALSE.

The ninth argument, nCharSet, specifies the font's character set. The available constants for this value are listed in Table 7.2.

TABLE 7.2. FONT CHARACTER SET CONSTANTS.

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

The system on which your application is running might have other character sets, and the OEM character set is system dependent, making it different for systems from different

manufacturers. If you are using one of these character sets, it is risky to try to manipulate the strings to be output, so it's best to just pass along the string to be displayed.

The tenth argument, nOutPrecision, specifies how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. The available values for this argument are

- OUT_CHARACTER_PRECIS
- OUT_DEFAULT_PRECIS
- OUT_DEVICE_PRECIS
- OUT_RASTER_PRECIS
- OUT_STRING_PRECIS
- OUT_STROKE_PRECIS
- OUT_TT_PRECIS

The OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS, and OUT_TT_PRECIS values control which font is chosen if there are multiple fonts with the same name. For instance, if you use the OUT_TT_PRECIS value and specify a font with both a TrueType and raster version, then the TrueType version is used. In fact, the OUT_TT_PRECIS value forces the system to use a TrueType font, even when the specified font does not have a TrueType version.

The eleventh argument, nClipPrecision, specifies how to clip characters that are partially outside of the display area. The values for this argument are

- CLIP_CHARACTER_PRECIS
- CLIP_DEFAULT_PRECIS
- CLIP_ENCAPSULATE
- CLIP_LH_ANGLES
- CLIP_MASK
- CLIP_STROKE_PRECIS
- CLIP_TT_ALWAYS

These values can be ORed together to specify a combination of clipping techniques.

The twelfth argument, nQuality, specifies the output quality and how carefully the GDI (Graphics Device Interface) must attempt to match the logical font attributes to the physical font output. The available values for this argument are

- DEFAULT_QUALITY
- DRAFT_QUALITY
- PROOF_QUALITY

The thirteenth argument, `nPitchAndFamily`, specifies the pitch and family of the font. This value consists of two values that are `OR`ed together to create a combination value. The first set of available values is

- `DEFAULT_PITCH`
- `VARIABLE_PITCH`
- `FIXED_PITCH`

This value specifies the pitch to be used with the font. The second set of available values specifies the family of fonts to be used. The available values for this portion of the argument are

- `FF_DECORATIVE`
- `FF_DONTCARE`
- `FF_MODERN`
- `FF_ROMAN`
- `FF_SCRIPT`
- `FF_SWISS`

The font family describes in a general way the appearance of a font. You can use the font family value to choose an alternative font when a specific font does not exist on a system. The final argument, `lpszFacename`, is a standard C-style string that contains the name of the font to be used. This font name comes from the font information received by the `EnumFontFamProc` callback function.

Using Fonts

Today you will build an application that allows the user to select from a list of available fonts to be displayed. The user will be able to enter some text to be displayed in the selected font, allowing the user to see what the font looks like.

Creating the Application Shell

To begin today's application, follow these steps:

1. Create a new project workspace using the MFC AppWizard. Name the project **`Day7`**.
2. Use the same defaults that you used for the previous day's projects, giving the application a title of **`Fonts`**.
3. Design the main dialog as in Figure 7.1, using the properties in Table 7.3.

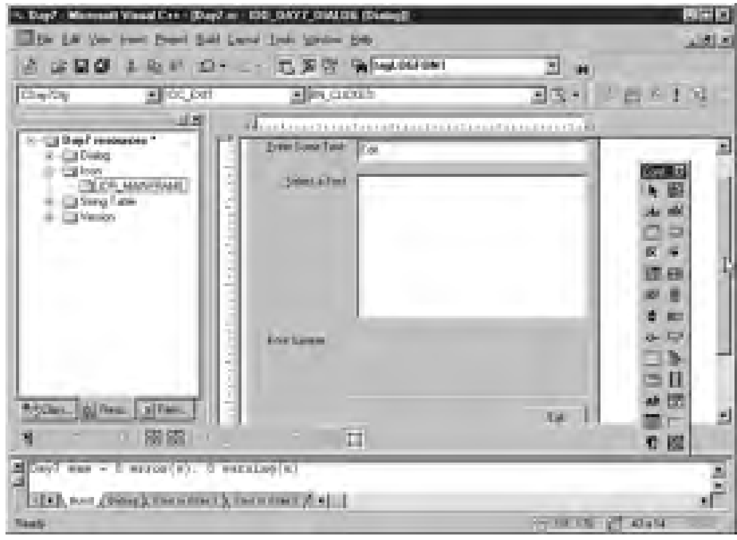


FIGURE 7.1.
The main dialog layout.

TABLE 7.3. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`&Enter Some Text:`
Edit Box	`ID`	`IDC_ESAMPTEXT`
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`&Select a Font`
List Box	`ID`	`IDC_LFONTS`
Group Box	`ID`	`IDC_STATIC`
	`Caption`	`Font Sample`
Static Text (inside group box; size to fill the group box)	`ID`	`IDC_DISPLAYTEXT`
	`Caption`	Empty string
Command Button	`ID`	`IDC_EXIT`
	`Caption`	`E&xit`

4. Using the Class Wizard, add the variables in Table 7.4 to the controls on the dialog.

TABLE 7.4. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
IDC_DISPLAYTEXT	m_ctlDisplayText	Control	CStatic
	m_strDisplayText	Value	CString
IDC_LFONTS	m_ctlFontList	Control	CListBox
	m_strFontName	Value	CString
IDC_ESAMPTEXT	m_strSampText	Value	CString

5. Attach a function to the `IDC_EXIT` button to close the application, as in the previous day's applications.

Building a List of Fonts

To be able to create your list of fonts, you need to add your callback function to get each font list and add it to the list box that you placed on the dialog window. To do this, edit the `Day7Dlg.h` header file and add the function declaration in Listing 7.1 near the top of the file. This function cannot be added through any of the tools available in Visual C++. You need to open the file and add it yourself.

LISTING 7.1. THE CALLBACK FUNCTION DECLARATION IN THE `Day7Dlg.h` HEADER FILE.

```
 1: #if _MSC_VER > 1000
 2: #pragma once
 3: #endif // _MSC_VER > 1000
 4:
 5: int CALLBACK EnumFontFamProc(LPENUMLOGFONT lpelf,
 6: LPNEWTEXTMETRIC lpntm, DWORD nFontType, long lParam);
 7:
 8: ///////////////////////////////////////////////////////////////////////////
 9: // CDay7Dlg dialog
10:
11: class CDay7Dlg : public CDialog
12: .
13: .
14: .
```

Once you add the function declaration to the header file, open the `Day7Dlg.cpp` source-code file, scroll to the bottom of the file, and add the function definition in Listing 7.2.

LISTING 7.2. THE CALLBACK FUNCTION DEFINITION IN THE `Day7Dlg.cpp` SOURCE FILE.

```
 1: int CALLBACK EnumFontFamProc(LPENUMLOGFONT lpelf,
 2: LPNEWTEXTMETRIC lpntm, DWORD nFontType, long lParam)
 3: {
 4:     // Create a pointer to the dialog window
 5:     CDay7Dlg* pWnd = (CDay7Dlg*) lParam;
 6:
 7:     // Add the font name to the list box
 8:     pWnd->m_ctlFontList.AddString(lpelf->elfLogFont.lfFaceName);
 9:     // Return 1 to continue font enumeration
10:     return 1;
11: }
```

Now that you have the callback function defined, you need to add a function to request the list of fonts from the operating system. To add this function, follow these steps:

1. Select the Class View tab on the project workspace pane.
2. Select the `CDay7Dlg` class, right-click the mouse, and select Add Member Function from the pop-up menu.
3. Specify the function type as `void`, the function declaration as `FillFontList`, and the access as `Private`. Click the OK button to close the dialog and add the function.
4. Edit the function definition as in Listing 7.3.

LISTING 7.3. THE `FillFontList` FUNCTION.

```
 1: void CDay7Dlg::FillFontList()
 2: {
 3:     LOGFONT lf;
 4:
 5:     // Initialize the LOGFONT structure
 6:     lf.lfCharSet = DEFAULT_CHARSET;
 7:     strcpy(lf.lfFaceName, "");
 8:     // Clear the list box
 9:     m_ctlFontList.ResetContent();
10:     // Create a device context variable
11:     CClientDC dc (this);
12:     // Enumerate the font families
13:     ::EnumFontFamiliesEx((HDC) dc, &lf,
14: (FONTENUMPROC) EnumFontFamProc, (LPARAM) this, 0);
15: }
```

5. Edit the `OnInitDialog` function to call the `FillFontList` function, as in Listing 7.4.

LISTING 7.4. THE EDITED `OnInitDialog` FUNCTION.

```
 1: BOOL CDay7Dlg::OnInitDialog()
 2: {
 3:     CDialog::OnInitDialog();
 4: .
 5: .
 6: .
 7:     // TODO: Add extra initialization here
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Fill the font list box
14:     FillFontList();
15:
16:     ///////////////////////
17:     // MY CODE ENDS HERE
18:     ///////////////////////
19:
20:     return TRUE;  // return TRUE  unless you set the focus to a control
21: }
```

If you compile and run your application now, you should find that your list box is filled with the names of all the fonts available on the system. However, there's one aspect of this list that you probably don't want in your application. Figure 7.2 shows many duplicate entries in the list of fonts in the list box. It would be nice if you could eliminate these duplicates and have only one line per font.

FIGURE 7.2.

Listing all the fonts in the system.

It turns out that the `EnumFontFamiliesEx` function call is synchronous in nature. This means that it doesn't return until all the fonts in the system are listed in calls to your

callback function. You can place code in the `FillFontList` function to remove all the duplicate entries once the list box is filled. To do this, modify the `FillFontList` function as in Listing 7.5.

LISTING 7.5. THE MODIFIED `FillFontList` FUNCTION.

```
 1: void CDay7Dlg::FillFontList()
 2: {
 3:     int iCount;            // The number of fonts
 4:     int iCurCount;        // The current font
 5:     CString strCurFont;        // The current font name
 6:     CString strPrevFont = "";    // The previous font name
 7:     LOGFONT lf;
 8:
 9:     // Initialize the LOGFONT structure
10:     lf.lfCharSet = DEFAULT_CHARSET;
11:     strcpy(lf.lfFaceName, "");
12:     // Clear the list box
13:     m_ctlFontList.ResetContent();
14:     // Create a device context variable
15:     CClientDC dc (this);
16:     // Enumerate the font families
17:     ::EnumFontFamiliesEx((HDC) dc, &lf,
18: (FONTENUMPROC) EnumFontFamProc, (LPARAM) this, 0);
19:     // Get the number of fonts in the list box
20:     iCount = m_ctlFontList.GetCount();
21:     // Loop from the last entry in the list box to the first,
22:     // searching for and deleting the duplicate entries
23:     for (iCurCount = iCount; iCurCount > 0; iCurCount--)
24:     {
25:         // Get the current font name
26:         m_ctlFontList.GetText((iCurCount - 1), strCurFont);
27:         // Is it the same as the previous font name?
28:         if (strCurFont == strPrevFont)
29:         {
30:             // If yes, then delete it
31:             m_ctlFontList.DeleteString((iCurCount - 1));
32:         }
33:         // Set the previous font name to the current font name
34:         strPrevFont = strCurFont;
35:     }
36: }
```

Notice that the `for` loop started at the end of the list and worked backward. This allowed you to delete the current entry without worrying about manipulating the loop counter to prevent skipping lines in the list box. If you compile and run your application, there shouldn't be any duplicate entries in the list of available fonts.

Setting the Font Sample Text

Before you can display the font for the user, you need to place some text into the display area. The edit box near the top of the dialog is where the user enters text to be displayed in the font selected. To add the functionality, do the following:

1. Edit the `OnInitDialog` function to add code to initialize the edit box and display text, as in Listing 7.6.

LISTING 7.6. THE MODIFIED `OnInitDialog` FUNCTION.

```
 1: BOOL CDay7Dlg::OnInitDialog()
 2: {
 3:     CDialog::OnInitDialog();
 4: .
 5: .
 6: .
 7:     // TODO: Add extra initialization here
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Fill the font list box
14:     FillFontList();
15:
16:     // Initialize the text to be entered
17:     m_strSampText = "Testing";
18:     // Copy the text to the font sample area
19:     m_strDisplayText = m_strSampText;
20:     // Update the dialog
21:     UpdateData(FALSE);
22:
23:     ///////////////////////
24:     // MY CODE ENDS HERE
25:     ///////////////////////
26:
27:     return TRUE;  // return TRUE  unless you set the focus to a control
28: }
```

2. Using the Class Wizard, add a function on the `EN_CHANGE` event message for the `IDC_ESAMPTEXT` edit box control.
3. Edit the function you just added, adding the code in Listing 7.7.

LISTING 7.7. THE `OnChangeEsamptext` FUNCTION.

```
 1: void CDay7Dlg::OnChangeEsamptext()
 2: {
 3:     // TODO: If this is a RICHEDIT control, the control will not
 4:     // send this notification unless you override the
        ➥CDialog::OnInitialUpdate()
 5:     // function and call CRichEditCrtl().SetEventMask()
 6:     // with the EN_CHANGE flag ORed into the mask.
 7:
 8:     // TODO: Add your control notification handler code here
 9:
10:     ///////////////////////
11:     // MY CODE STARTS HERE
12:     ///////////////////////
13:
14:     // Update the variables with the dialog controls
15:     UpdateData(TRUE);
16:
17:     // Copy the current text to the font sample
18:     m_strDisplayText = m_strSampText;
19:
20:     // Update the dialog with the variables
21:     UpdateData(FALSE);
22:
23:     ///////////////////////
24:     // MY CODE ENDS HERE
25:     ///////////////////////
26: }
```

If you compile and run your application, you should be able to type text into the edit box and see it change in the font display area in the group box below.

Selecting a Font to Display

Before you can start changing the font for the display area, you'll need to have a `CFont` member variable of the dialog class that you can use to set and change the display font. To add this variable, follow these steps:

1. In the Class View of the workspace pane, right-click the mouse on the `CDay7Dlg` class. Select Add Member Variable from the pop-up menu.
2. Specify the variable type as `CFont`, the variable name as `m_fSampFont`, and the access as `Private`. Click the OK button to close the dialog box and add the variable.

When adding the code to use the selected font, you'll add it as a separate function that is not attached to a control. Why you do this will become clear as you proceed further

through building and running today's application. To add the function to display and use the selected font, follow these steps:

1. In the Class View of the workspace pane, right-click the mouse on the `CDay7Dlg` class. Select Add Member Function from the pop-up menu.
2. Specify the function type as `void`, the function declaration as `SetMyFont`, and the access as `Private`. Click the OK button to close the dialog and add the function.
3. Edit the function, adding the code in Listing 7.8.

LISTING 7.8. THE `SetMyFont` FUNCTION.

```
 1: void CDay7Dlg::SetMyFont()
 2: {
 3:     CRect rRect;        // The rectangle of the display area
 4:     int iHeight;    // The height of the display area
 5:
 6:     // Has a font been selected?
 7:     if (m_strFontName != "")
 8:     {
 9:         // Get the dimensions of the font sample display area
10:         m_ctlDisplayText.GetWindowRect(&rRect);
11:         // Calculate the area height
12:         iHeight = rRect.top - rRect.bottom;
13:         // Make sure the height is positive
14:         if (iHeight < 0)
15:             iHeight = 0 - iHeight;
16:         // Release the current font
17:         m_fSampFont.Detach();
18:         // Create the font to be used
19:         m_fSampFont.CreateFont((iHeight - 5), 0, 0, 0, FW_NORMAL,
20:                0, 0, 0, DEFAULT_CHARSET, OUT_CHARACTER_PRECIS,
21:                CLIP_CHARACTER_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH ¦
22:                FF_DONTCARE, m_strFontName);
23:
24:         // Set the font for the sample display area
25:         m_ctlDisplayText.SetFont(&m_fSampFont);
26:     }
27: }
```

4. Using the Class Wizard, add a function to the `LBN_SELCHANGE` event message for the `IDC_LFONTS` list box. Edit the function, adding the code in Listing 7.9.

LISTING 7.9. THE `OnSelchangeLfonts` FUNCTION.

```
 1: void CDay7Dlg::OnSelchangeLfonts()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Update the variables with the dialog controls
10:     UpdateData(TRUE);
11:
12:     // Set the font for the sample
13:     SetMyFont();
14:
15:     ///////////////////////
16:     // MY CODE ENDS HERE
17:     ///////////////////////
18: }
```

In the `SetMyFont` function, you first checked to make sure that a font had been selected. Next, you retrieved the area of the static text control that will be used to display the font. This enables you to specify a font height just slightly smaller than the height of the area you have available to display the font in. After you calculated the height of the static text control and made sure that it is a positive value, you created the selected font and told the static text control to use the newly created font.

In the `OnSelchangeLfonts` function, you copy the control values to the attached variables and then call the `SetMyFont` function to use the selected font. If you compile and run your application, you should be able to select a font and see it displayed in the sample static text control, as in Figure 7.3.

FIGURE 7.3.
Displaying the selected font.

Summary

Today you learned how to use fonts in Visual C++ applications. You learned how to get a list of the available fonts that are loaded on the system and then how to create a font for use on a display object. You learned how you can create and use callback functions to get a list of resources from the Windows operating system. You also learned how you can access controls from the callback function using a window pointer that you passed to the function requesting the resource list.

Q&A

Q The `CreateFont` function has a lot of arguments to specify and pass. Is there any other alternative to using this function?

A Yes, there is, although you still specify all of the same information. A structure called `LOGFONT` contains all the same attributes that are passed to the `CreateFont` function. You can declare an instance of this structure, initializing the attributes to default values, and then pass this structure to the `CreateFontIndirect` function. If you make numerous font changes, this approach is preferable because you could use the same instance of the structure, modifying those attributes that are changing from the current settings and using it to create the various fonts.

The way that you use this alternative way of creating the font is to declare an instance of the `LOGFONT` structure as a member of the dialog class and then initialize all the attributes before calling the `SetMyFont` function. In the `SetMyFont` function, you modify it as shown in Listing 7.10.

LISTING 7.10. THE MODIFIED `SetMyFont` FUNCTION.

```
 1: void CDay7Dlg::SetMyFont()
 2: {
 3:
 4:     // Has a font been selected?
 5:     if (m_strFontName != "")
 6:     {
 7:         // Assume that the font size has already been initialized in the
 8:         // m_lLogFont structure. This allows you to only have to specify
 9:         // the font name.
10:         tcscpy(m_lLogFont.lfFaceName, m_strFontName);
11:         // Create the font to be used
12:         m_fSampFont.CreateFontIndirect(&m_lLogFont);
13:
14:         // Set the font for the sample display area
15:         m_ctlDisplayText.SetFont(&m_fSampFont);
16:     }
17: }
```

Q How can I limit the fonts in my list to just the TrueType fonts?

A You can check the `nFontType` argument to your callback function to determine the font type. For instance, if you want to include only TrueType fonts in your list of fonts, you modify your callback function to mask the `nFontType` argument with the `TRUETYPE_FONTTYPE` constant and check to see if the resulting value equals the `TRUETYPE_FONTTYPE` value, as in the following:

```
int CALLBACK EnumFontFamProc(LPENUMLOGFONT lpelf,
LPNEWTEXTMETRIC lpntm, DWORD nFontType, long lParam)
{
    // Create a pointer to the dialog window
    CDay7Dlg* pWnd = (CDay7Dlg*) lParam;

    // Limit the list to TrueType fonts
    if ((nFontType & TRUETYPE_FONTTYPE) == TRUETYPE_FONTTYPE)
    {
        // Add the font name to the list box
        pWnd->m_ctlFontList.AddString(
                        lpelf->elfLogFont.lfFaceName);
    }
    // Return 1 to continue font enumeration
    return 1;
}
```

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. How can you specify that the text is to be underlined?
2. How can you print your text upside down?
3. How many times is the `EnumFontFamProc` callback function called by the operating system?

Exercises

1. Add a check box to switch between using the entered text to display the font and using the font name to display the font, as in Figure 7.4.

FIGURE 7.4.
Displaying the selected font with the font name.

2. Add a check box to display the font sample in italics, as in Figure 7.5.

FIGURE 7.5.
Displaying the selected font in italics.

WEEK 1

In Review

Well, you've made it through the first week. By this point, you've gotten a good taste for what's possible when building applications with Visual C++. Now it's time to look back over what's been covered and what you should have learned up to this point.

What you might want to do at this point, to cement your understanding of how you can use these elements in your own applications, is to try designing and building a couple of simple applications of your own. You can use a variety of controls and add some additional dialogs, just so you can make sure that you do understand and are comfortable with these topics. In fact, you might want to try out all the topics that I've covered up to this point in small applications of your own design. That's the true test of your understanding of how the concepts work. You might also want to dive into the MFC documentation to learn a little about some of the more advanced functionality that I haven't covered to see if you can figure out how you can use and incorporate it into your applications.

One of the most important things that you should understand at this point is how you can use controls and dialog windows in your applications to get and display information to the user. This is an important part of any Windows application because just about every application interacts with the user in some way. You should be able to place any of the standard controls on a dialog in your application and be able to incorporate them into your application without any problem. Likewise, you should be comfortable with using the standard message box and dialog windows provided to your application by the Windows operating system. You should also be

able to create and incorporate your own custom dialog windows into any application you might want to build. If you don't feel comfortable with any of these topics, you might want to go back and review Day 2 to get a better understanding of how you can use controls and Day 5 to understand how you can incorporate standard and custom dialog windows into your applications.

Another key skill that you will be using in the majority of your applications is the ability to build and incorporate menus into your applications. You need to have a firm understanding of how to design a good menu, how to make sure that there are no conflicting mnemonics, and how you can attach application functionality to the menu selections. At this point, you should be able to create your own customized menus, with entries for each of the various functions that your application performs, and integrate it with your application with no problems. If you aren't 100% comfortable with this topic, you might want to go back and study Day 6 a little more.

You will find that there are various situations in which you need to have some means of triggering actions on a regular basis or in which you need to keep track of how long some process has been running. For both of these situations, as well as numerous others, you'll often find yourself turning to the use of timers in your application. If you are even slightly foggy on how you can integrate timers into your applications, you will definitely want to go back and review Day 4.

Understanding how you can use text and fonts in your applications will allow you to build more flexibility into the appearance of your applications—to give your users the ability to customize the appearance as they want. You will be able to examine the available fonts on the computer on which your application is running and, if a font that you want to use in your application isn't available, choose another font that is close to use instead. If you still have any questions on how the font infrastructure in Windows works and how you can use it in your applications, you'll want to go back and review Day 7 once more.

Depending on the nature of your application, being able to capture and track mouse and keyboard actions by the user can be very important. If you are building a drawing application, this is crucial information. If you are building an application that needs to include drag-and-drop capabilities, this is important once again. There are any number of situations in which you'll want to include this functionality into your applications. By this point, you should understand how you can capture the various mouse events and determine which mouse buttons are involved in the event. You should also be able to capture keyboard events in situations where the keyboard input isn't captured by any controls that are on the window. If you don't feel like you have a complete grasp of this, you should take another look at Day 3.

Finally, you should be familiar with the Visual C++ development environment, the Developer Studio. You should have a good understanding of what each area of the environment is for and how you can use the various tools and utilities in building your applications. You should be comfortable with using the workspace pane to navigate around your application project, locating and bringing into the various editors and designers any part of your application. You should be comfortable with locating and redesigning the icon that will be displayed to represent your application and with finding any member functions or variables in any of your application's classes.

By now you should be getting fairly comfortable working with Visual C++. If you feel like you understand all the topics that I've covered so far, you are ready to continue forward, learning more about the various things that you can do, and functionality that you can build, using Visual C++ as your programming tool. With that said, it's on to the second week…

WEEK 2

At a Glance

In the second week, you'll dive into several more involved topics. These topics are still very much core to building Windows applications. You'll find yourself using what you learn in this week, along with what you learned during the first week, in just about all the applications that you build with Visual C++.

To start the week, on Day 8, you'll learn how to draw graphics in a Windows application. You'll learn how to draw simple lines, rectangles, and ellipses. What's more important—you'll learn about the device context and how you can use it to draw your graphics without worrying about the graphics hardware your users might or might not have in their computers.

On Day 9, you'll learn how easy it is to incorporate ActiveX controls into your applications. You'll see how Visual C++ builds custom C++ classes around the controls that you add to your project, enabling you to interact with an added control just as if it were another C++ object.

On Day 10, you'll learn how to build a basic Single Document Interface (SDI) application. You'll learn about the Document/View architecture that is used with Visual C++ for building this style of application, and you'll learn how you can use it to build your own applications.

On Day 11, you'll learn how you can apply what you learned about building SDI applications to building Multiple Document Interface (MDI) applications. You'll see how you can use the same Document/View architecture to create MDI applications, some of the most common style of Windows applications available today.

On Day 12, you'll learn how you can create and modify your own toolbars and status bars. You'll learn how you can attach toolbar buttons to menus in your application and how you can add additional toolbars. You'll also learn how you can place your own informational elements on the status bar at the bottom of most Windows applications and how you can keep the status bar updated with the status of your application.

On Day 13, you'll see how you can use the structure provided for you by the Document/View architecture to save and restore the data created in your application. You'll learn how flexible this facility is and how you can store different data types in the same file, restoring them to your application just as they were when you first saved them.

Finally, rounding out the week on Day 14, you'll learn how easy it is to build a database application with an ODBC database. You'll learn how to query a set of records from the database and how to allow the user to edit and modify them, saving the changes back to the database.

When you finish this week, you'll be well prepared for tackling most basic application development tasks with Visual C++. You might want to take a short break at that point to experiment a bit—trying to build various types of applications, pushing your skills, and learning what your limits are (and aren't)—before jumping into the final week of more advanced topics.

WEEK 2

DAY 8

Adding Flash—Incorporating Graphics, Drawing, and Bitmaps

You've probably noticed that a large number of applications use graphics and display images. This adds a certain level of flash and polish to the application. With some applications, graphics are an integral part of their functionality. Having a good understanding of what's involved in adding these capabilities to your applications is a key part of programming for the Windows platform. You've already learned how you can draw lines and how you can string a series of these lines together to make a continuous drawing. Today, you're going to go beyond that capacity and learn how you can add more advanced graphics capabilities to your applications. Today, you will learn

- How Windows uses a device context to translate drawing instructions into graphics output.
- How you can determine the level of control you have over the graphics output through different mapping modes.

- How Windows uses pens and brushes to draw different portions of the graphics image.
- How you can load and display bitmaps dynamically.

Understanding the Graphics Device Interface

The Windows operating system provides you with a couple of levels of abstraction for creating and using graphics in your applications. During the days of DOS programming, you needed to exercise a great deal of control over the graphics hardware to draw any kind of images in an application. This control required an extensive knowledge and understanding of the various types of graphics cards that users might have in their computers, along with their options for monitors and resolutions. There were a few graphics libraries that you could buy for your applications, but overall, it was fairly strenuous programming to add this capability to your applications.

With Windows, Microsoft has made the job much easier. First, Microsoft provides you with a virtual graphics device for all of your Windows applications. This virtual device doesn't change with the hardware but remains the same for all possible graphics hardware that the user might have. This consistency provides you with the ability to create whatever kind of graphics you want in your applications because you know that the task of converting them to something that the hardware understands isn't your problem.

Device Contexts

Before you can create any graphics, you must have the device context in which the graphics will be displayed. The device context contains information about the system, the application, and the window in which you are drawing any graphics. The operating system uses the device context to learn in which context a graphic is being drawn, how much of the area is visible, and where on the screen it is currently located.

When you draw graphics, you always draw them in the context of an application window. At any time, this window may be full view, minimized, partly hidden, or completely hidden. This status is not your concern because you draw your graphics on the window using its device context. Windows keeps track of each device context and uses it to determine how much and what part of the graphics you draw to actually display for the user. In essence, the device context you use to display your graphics is the visual context of the window in which you draw them.

The device context uses two resources to perform most of its drawing and graphics functions. These two resources are pens and brushes. Much like their real-world counterparts, pens and brushes perform similar yet different tasks. The device context uses pens to

draw lines and shapes, whereas brushes paint areas of the screen. It's the same idea as working on paper when you use a pen to draw an outline of an image and then pick up a paintbrush to fill in the color between the lines.

The Device Context Class

In Visual C++, the MFC device context class (CDC) provides numerous drawing functions for drawing circles, squares, lines, curves, and so on. All these functions are part of the device context class because they all use the device context information to draw on your application windows.

You create a device context class instance with a pointer to the window class that you want to associate with the device context. This allows the device context class to place all of the code associated with allocating and freeing a device context in the class constructor and destructors.

Device context objects, as well as all of the various drawing objects, are classified as resources in the Windows operating system. The operating system has only a limited amount of these resources. Although the total number of resources is large in recent versions of Windows, it is still possible to run out of resources if an application allocates them and doesn't free them correctly. This loss is known as a resource leak, and much like a memory leak, it can eventually lock up a user's system. As a result, it's advisable to create these resources in the functions where they will be used and then delete them as soon as you are finished with them.

Following this advised approach to using device contexts and their drawing resources, you use them almost exclusively as local variables within a single function. The only real exception is when the device context object is created by Windows and passed into the event-processing function as an argument.

The Pen Class

You have already seen how you can use the pen class, `CPen`, to specify the color and width for drawing lines onscreen. `CPen` is the primary resource tool for drawing any kind of line onscreen. When you create an instance of the `CPen` class, you can specify the line type, color, and thickness. After you create a pen, you can select it as the current drawing tool for the device context so that it is used for all of your drawing commands to the

device context. To create a new pen, and then select it as the current drawing pen, you use the following code:

```
// Create the device context
CDC dc(this);
// Create the pen
CPen lPen(PS_SOLID, 1, RGB(0, 0, 0));
// Select the pen as the current drawing pen
dc.SelectObject(&lPen);
```

You can use a number of different pen styles. These pen styles all draw different patterns when drawing lines. Figure 8.1 shows the basic styles that can be used in your applications with any color.

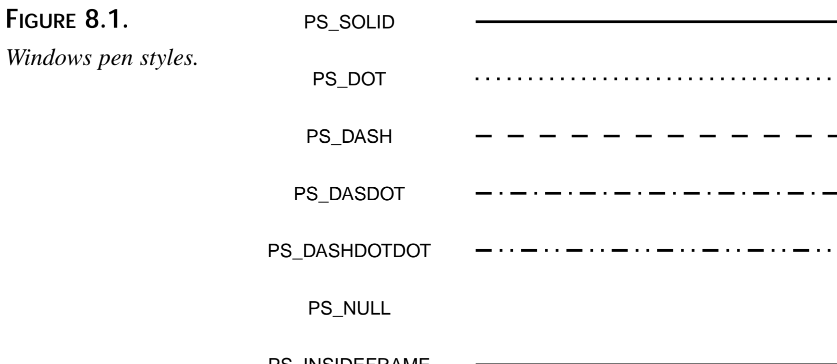


FIGURE 8.1.
Windows pen styles.

When you use any of these line styles with a pen thickness greater than 1, all of the lines appear as solid lines. If you want to use any line style other than `PS_SOLID`, you need to use a pen width of 1.

Along with the line style that the pen should draw, you also have to specify the pen's width and color. The combination of these three variables specifies the appearance of the resulting lines. The line width can range from 1 on up, although when you reach a width of 32, it's difficult to exercise any level of precision in your drawing efforts.

You specify the color as a RGB value, which has three separate values for the brightness of the red, green, and blue color components of the pixels on the computer screen. These three separate values can range from 0 to 255, and the `RGB` function combines them into a single value in the format needed by Windows. Some of the more common colors are listed in Table 8.1.

TABLE 8.1. COMMON WINDOWS COLORS.

Color	*Red*	*Green*	*Blue*
Black	0	0	0
Blue	0	0	255
Dark blue	0	0	128
Green	0	255	0
Dark green	0	128	0
Cyan	0	255	255
Dark cyan	0	128	128
Red	255	0	0
Dark red	128	0	0
Magenta	255	0	255
Dark magenta	128	0	128
Yellow	255	255	0
Dark yellow	128	128	0
Dark gray	128	128	128
Light gray	192	192	192
White	255	255	255

The Brush Class

The brush class, `CBrush`, allows you to create brushes that define how areas will be filled in. When you draw shapes that enclose an area and fill in the enclosed area, the outline is drawn with the current pen, and the interior of the area is filled by the current brush. Brushes can be solid colors (specified using the same RGB values as with the pens), a pattern of lines, or even a repeated pattern created from a small bitmap. If you want to create a solid-color brush, you need to specify the color to use:

```
CBrush lSolidBrush(RGB(255, 0, 0));
```

To create a pattern brush, you need to specify not only the color but also the pattern to use:

```
CBrush lPatternBrush(HS_BDIAGONAL, RGB(0, 0, 255));
```

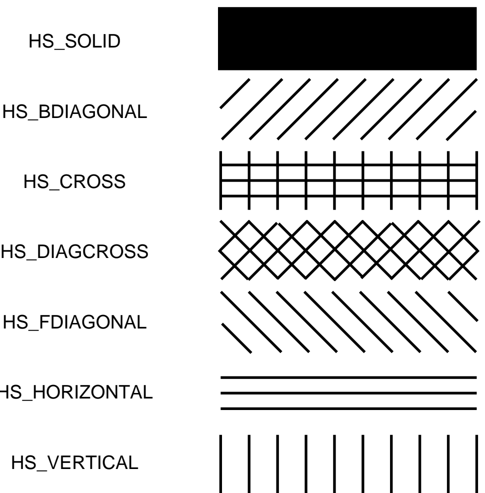
After you create a brush, you can select it with the device context object, just like you do with pens. When you select a brush, it is used as the current brush whenever you draw something that uses a brush.

As with pens, you can select a number of standard patterns when creating a brush, as shown in Figure 8.2. In addition to these patterns, an additional style of brush, `HS_BITMAP`, uses a bitmap as the pattern for filling the specified area. This bitmap is limited in size to 8 pixels by 8 pixels, which is a smaller bitmap than normally used for toolbars and other small images. If you supply it with a larger bitmap, it takes only the upper-left corner, limiting it to an 8-by-8 square. You can create a bitmap brush by creating a bitmap resource for your application and assigning it an object ID. After you do this, you can create a brush with it by using the following code:

```
CBitmap m_bmpBitmap;

// Load the image
m_bmpBitmap.LoadBitmap(IDB_MYBITMAP);
// Create the brush
CBrush lBitmapBrush(&m_bmpBitmap);
```

FIGURE 8.2.
Standard brush patterns.



If you want to create your own custom pattern for use as a brush, you can create the pattern as an 8-by-8 bitmap and use the bitmap brush. This allows you to extend the number of brush patterns far beyond the limited number of standard patterns.

The Bitmap Class

When you want to display images in your applications, you have a couple of options. You can add fixed bitmaps to your application, as resources with object IDs assigned to

them and use static picture controls or an ActiveX control that displays images. You can also use the bitmap class, `CBitmap`, to exercise complete control over the image display. If you use the bitmap class, you can dynamically load bitmap images from files on the system disk, resizing the images as necessary to make them fit in the space you've allotted.

If you add the bitmap as a resource, you can create an instance of the `CBitmap` class using the resource ID of the bitmap as the image to be loaded. If you want to load a bitmap from a file, you can use the `LoadImage` API call to load the bitmap from the file. After you load the bitmap, you can use the handle for the image to attach the image to the `CBitmap` class, as follows:

```
// Load the bitmap file
HBITMAP hBitmap = (HBITMAP)::LoadImage(AfxGetInstanceHandle(),
                    m_sFileName, IMAGE_BITMAP, 0, 0,
                    LR_LOADFROMFILE ¦ LR_CREATEDIBSECTION);
// Attach the loaded image to the CBitmap object.
m_bmpBitmap.Attach(hBitmap);
```

After you load the bitmap into the `CBitmap` object, you can create a second device context and select the bitmap into it. When you've created the second device context, you need to make it compatible with the primary device context before the bitmap is selected into it. Because device contexts are created by the operating system for a specific output device (screen, printer, and so on), you have to make sure that the second device context is also attached to the same output device as the first.

```
// Create a device context
CDC dcMem;
// Make the new device context compatible with the real DC
dcMem.CreateCompatibleDC(dc);
// Select the bitmap into the new DC
dcMem.SelectObject(&m_bmpBitmap);
```

When you select the bitmap into a compatible device context, you can copy the bitmap into the regular display device context using the `BitBlt` function:

```
// Copy the bitmap to the display DC
dc->BitBlt(10, 10, bm.bmWidth,
                 bm.bmHeight, &dcMem, 0, 0,
                 SRCCOPY);
```

You can also copy and resize the image using the `StretchBlt` function:

```
// Resize the bitmap while copying it to the display DC
dc->StretchBlt(10, 10, (lRect.Width() - 20),
                (lRect.Height() - 20), &dcMem, 0, 0,
                 bm.bmWidth, bm.bmHeight, SRCCOPY);
```

By using the `StretchBlt` function, you can resize the bitmap so that it will fit in any area on the screen.

Mapping Modes and Coordinate Systems

When you are preparing to draw some graphics on a window, you can exercise a lot of control over the scale you are using and the area in which you can draw. You can control these factors by specifying the mapping mode and the drawing area.

By specifying the mapping mode, you can control how the coordinates that you specify are translated into locations on the screen. The different mapping modes translate each point into a different distance. You can set the mapping mode by using the `SetMapMode` device context function:

```
dc->SetMapMode(MM_ANSIOTROPIC);
```

The available mapping modes are listed in Table 8.2.

TABLE 8.2. MAPPING MODES.

Mode	*Description*
`MM_ANSIOTROPIC`	Logical units are converted into arbitrary units with arbitrary axes.
`MM_HIENGLISH`	Each logical unit is converted into 0.001 inch. Positive x is to the right, and positive y is up.
`MM_HIMETRIC`	Each logical unit is converted into 0.01 millimeter. Positive x is to the right, and positive y is up.
`MM_ISOTROPIC`	Logical units are converted into arbitrary units with equally scaled axes.
`MM_LOENGLISH`	Each logical unit is converted into 0.01 inch. Positive x is to the right, and positive y is up.
`MM_LOMETRIC`	Each logical unit is converted into 0.1 millimeter. Positive x is to the right, and positive y is up.
`MM_TEXT`	Each logical unit is converted into 1 pixel. Positive x is to the right, and positive y is down.
`MM_TWIPS`	Each logical unit is converted into 1/20 of a point (approximately 1/1440 inch). Positive x is to the right, and positive y is up.

If you use either the `MM_ANSIOTROPIC` or `MM_ISOTROPIC` mapping modes, you can use either the `SetWindowExt` or `SetViewportExt` functions to specify the drawing area where your graphics should appear.

Creating a Graphics Application

To get a good understanding of how you can put all of this information to use, you'll build an application that incorporates a lot of what I've covered so far today. This application will have two independent windows, one with a number of options to choose for the shape, tool, and color to be displayed. The other window will act as a canvas, where all of the selected options will be drawn. The user can select whether to display lines, squares, circles, or a bitmap on the second window. The user can also specify the color and choose whether to display the pen or brush for the circles and squares.

Generating the Application Shell

As you have learned by now, the first step in building an application is generating the initial application shell. This shell provides the basic application functionality, displaying your first application dialog, along with all startup and shutdown functionality.

For the application that you will build today, you need to start with a standard dialog-style application shell. You can create this for your application by starting a new AppWizard project, providing a suitable project name, such as `Graphics`. After you are in the AppWizard, specify that you are creating a dialog-style application. At this point, you can accept all of the default settings, although you won't need ActiveX support, and you can specify a more descriptive window title if you want.

Designing the Main Dialog

After you make your way through the AppWizard, you're ready to start designing your primary dialog. This window will contain three groups of radio buttons: one group for specifying the drawing tool, the next to specify the drawing shape, and the third to specify the color. Along with these groups of radio buttons, you'll have two buttons on the window: one to open a File Open dialog, for selecting a bitmap to be displayed, and the other to close the application.

To add all these controls to your dialog, lay them out as shown in Figure 8.3 and specify the control properties listed in Table 8.3.

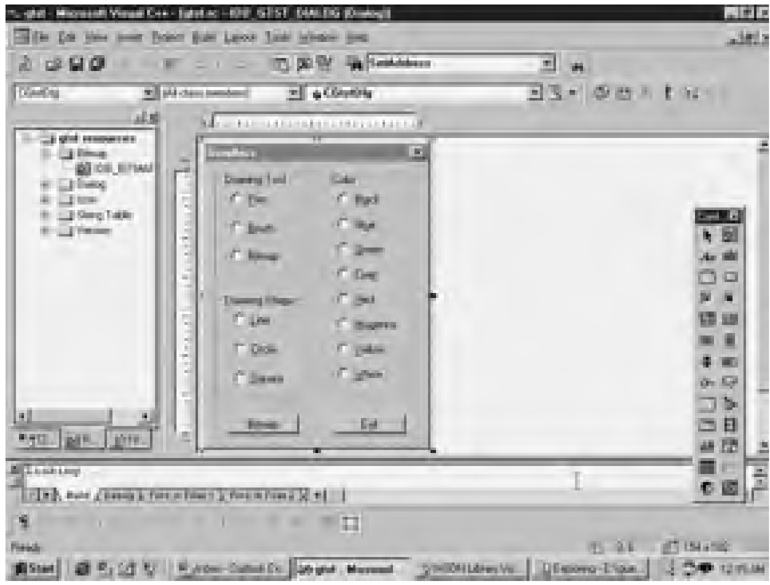


FIGURE 8.3. *The main dialog layout.*

TABLE 8.3. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Group Box	ID	`IDC_STATIC`
	Caption	`Drawing Tool`
Radio Button	ID	`IDC_RTPEN`
	Caption	`&Pen`
	Group	`Checked`
Radio Button	ID	`IDC_RTBRUSH`
	Caption	`&Brush`
Radio Button	ID	`IDC_RTBITMAP`
	Caption	`B&itmap`
Group Box	ID	`IDC_STATIC`
	Caption	`Drawing Shape`
Radio Button	ID	`IDC_RSLINE`
	Caption	`&Line`
	Group	`Checked`
Radio Button	ID	`IDC_RSCIRCLE`
	Caption	`&Circle`

Object	*Property*	*Setting*
Radio Button	ID	`IDC_RSSQUARE`
	Caption	`&Square`
Group Box	ID	`IDC_STATIC`
	Caption	`Color`
Radio Button	ID	`IDC_RCBLACK`
	Caption	`Bl&ack`
	Group	`Checked`
Radio Button	ID	`IDC_RCBLUE`
	Caption	`Bl&ue`
Radio Button	ID	`IDC_RCGREEN`
	Caption	`&Green`
Radio Button	ID	`IDC_RCCYAN`
	Caption	`Cya&n`
Radio Button	ID	`IDC_RCRED`
	Caption	`&Red`
Radio Button	ID	`IDC_RCMAGENTA`
	Caption	`&Magenta`
Radio Button	ID	`IDC_RCYELLOW`
	Caption	`&Yellow`
Radio Button	ID	`IDC_RCWHITE`
	Caption	`&White`
Command Button	ID	`IDC_BBITMAP`
	Caption	`Bi&tmap`
Command Button	ID	`IDC_BEXIT`
	Caption	`E&xit`

When you finish designing your main dialog, you need to assign one variable to each of the groups of radio buttons. To do this, open the Class Wizard and assign one integer variable to each of the three radio button object IDs there. Remember that only the object IDs for the radio buttons with the Group option checked will appear in the Class Wizard. All of the radio buttons that follow will be assigned to the same variable, with sequential values, in the order of the object ID values. For this reason, it is important to create all of the radio buttons in each group in the order that you want their values to be sequenced.

To assign the necessary variables to the radio button groups in your application, open the Class Wizard and add the variables in Table 8.4 to the objects in your dialog.

TABLE 8.4. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
IDC_RTPEN	m_iTool	Value	int
IDC_RSLINE	m_iShape	Value	int
IDC_RCBLACK	m_iColor	Value	int

While you have the Class Wizard open, you might want to switch back to the first tab and add an event-handler function to the Exit button, calling the `OnOK` function in the code for this button. You can compile and run your application now, making sure that you have all of the radio button groups defined correctly, that you can't select two or more buttons in any one group, and that you can select one button in each group without affecting either of the other two groups.

Adding the Second Dialog

When you design the main dialog, you'll add the second window that you'll use as a canvas to paint your graphics on. This dialog will be a modeless dialog, which will remain open the entire time the application is running. You will put no controls on the dialog, providing a clean canvas for drawing.

To create this second dialog, go to the Resources tab in the workspace pane. Right-click the Dialogs folder in the resource tree. Select Insert Dialog from the pop-up menu. When the new dialog is open in the window designer, remove all of the controls from the window. After you remove all of the controls, open the properties dialog for the window and uncheck the System Menu option on the second tab of properties. This will prevent the user from closing this dialog without exiting the application. You'll also want to give this dialog window an object ID that will describe its function, such as `IDD_PAINT_DLG`.

After you finish designing the second dialog, create a new class for this window by opening the Class Wizard. When you try to open the Class Wizard, you'll be asked if you want to create a new class for the second dialog window. Leave this option at its default setting and click the OK button. When asked to specify the name of the new class on the next dialog, give the class a suitable name, such as `CPaintDlg`, and be sure that the base class is set to `CDialog`. After you click OK on this dialog and create the new class, you can close the Class Wizard.

> You need to make sure that the new dialog is selected when you try to open the Class Wizard. If the dialog is not selected, and you've switched to another object, or even some code in your application, the Class Wizard will not know that you need a class for the second dialog in your application.

Now that you have the second dialog defined, you need to add the code in the first dialog window to open the second dialog. You can accomplish this by adding two lines of code to the `OnInitDialog` function in the first window's class. First, create the dialog using the `Create` method of the `CDialog` class. This function takes two arguments: the object ID of the dialog and a pointer to the parent window, which will be the main dialog. The second function will be the `ShowWindow` function, passing the value `SW_SHOW` as the only argument. This function displays the second dialog next to the first dialog. Add a couple of lines of variable initialization to make your `OnInitDialog` function resemble Listing 8.1.

LISTING 8.1. THE `OnInitDialog` FUNCTION.

```
1:  BOOL CGraphicsDlg::OnInitDialog()
2:  {
3:      CDialog::OnInitDialog();
4:
.
.
.
27:
28:     // TODO: Add extra initialization here
29:
30:     ///////////////////////
31:     // MY CODE STARTS HERE
32:     ///////////////////////
33:
34:     // Initialize the variables and update the dialog window
35:     m_iColor = 0;
36:     m_iShape = 0;
37:     m_iTool = 0;
38:     UpdateData(FALSE);
39:
40:     // Create the second dialog window
41:     m_dlgPaint.Create(IDD_PAINT_DLG, this);
42:     // Show the second dialog window
43:     m_dlgPaint.ShowWindow(SW_SHOW);
```

continues

LISTING 8.1. CONTINUED

```
44:
45:     ///////////////////////
46:     // MY CODE ENDS HERE
47:     ///////////////////////
48:
49:     return TRUE;  // return TRUE  unless you set the focus to a control
50: }
```

Before you can compile and run your application, you'll need to include the header for the second dialog class in the source code for the first dialog. You'll also need to add the second dialog class as a variable to the first—which is a simple matter of adding a member variable to the first dialog class, specifying the variable type as the class type, in this case `CPaintDlg`, giving the variable the name that you used in Listing 8.1, `m_dlgPaint`, and specifying the variable access as private. To include the header file in the first dialog, scroll to the top of the source code for the first dialog and add an include statement, as in Listing 8.2.

LISTING 8.2. THE INCLUDE STATEMENT OF THE MAIN DIALOG.

```
1: // GraphicsDlg.cpp : implementation file
2: //
3:
4: #include "stdafx.h"
5: #include "Graphics.h"
6: #include "PaintDlg.h"
7: #include "GraphicsDlg.h"
8:
```

Conversely, you'll need to include the header file for the main dialog in the source code for the second dialog. You can edit this file, `PaintDlg.cpp`, making the include statements match those in Listing 8.2.

If you compile and run your application, you should see your second dialog window open along with the first window. What you'll also noticed is that when you close the first dialog, and thus close the application, the second dialog window also closes, even though you didn't add any code to make this happen. The second dialog is a child window to the first dialog. When you created the second dialog, on line 41 of the code listing, you passed a pointer to the first dialog window as the parent window for the second window. This set up a parent-child relationship between these two windows. When the

parent closes, so does the child. This is the same relationship the first dialog window has with all of the controls you placed on it. Each of those controls is a child window of the dialog. In a sense, what you've done is make the second dialog just another control on the first dialog.

Adding the Graphics Capabilities

Because all of the radio button variables are declared as public, the second dialog will be able to see and reference them as it needs to. You can place all of the graphic drawing functionality into the second dialog class. However, you do need to place some functionality into the first dialog to keep the variables synchronized and to tell the second dialog to draw its graphics. Accomplishing this is simpler than you might think.

Whenever a window needs to be redrawn (it may have been hidden behind another window and come to the front or minimized or off the visible screen and now in view), the operating system triggers the dialog's `OnPaint` function. You can place all the functionality for drawing your graphics in this function and make persistent the graphics you display.

Now that you know where to place your code to display the graphics, how can you cause the second dialog to call its `OnPaint` function whenever the user changes one of the selections on the first dialog? Well, you could hide and then show the second dialog, but that might look a little peculiar to the user. Actually, a single function will convince the second window that it needs to redraw its entire dialog. This function, `Invalidate`, requires no arguments and is a member function of the `CWnd` class, so it can be used on any window or control. The `Invalidate` function tells the window, and the operating system, that the display area of the window is no longer valid and that it needs to be redrawn. You can trigger the `OnPaint` function in the second dialog at will, without resorting to any awkward tricks or hacks.

At this point, we have determined that all of the radio buttons can use the same functionality on their clicked events. You can set up a single event-handler function for the clicked event on all of the radio button controls. In this event function, you'll need to synchronize the class variables with the dialog controls by calling the `UpdateData` function and then tell the second dialog to redraw itself by calling its `Invalidate` function. You can write a single event handler that does these two things with the code in Listing 8.3.

LISTING 8.3. THE `OnRSelection` FUNCTION.

```
1: void CGraphicsDlg::OnRSelection()
2: {
3:     // TODO: Add your control notification handler code here
4:
5:     // Synchronize the data
6:     UpdateData(TRUE);
7:     // Repaint the second dialog
8:     m_dlgPaint.Invalidate();
9: }
```

Drawing Lines

You can compile and run your application at this point, and the second dialog redraws itself whenever you choose a different radio button on the main dialog, but you wouldn't notice anything happening. At this point, you are triggering the redraws, but you haven't told the second dialog what to draw, which is the next step in building this application.

The easiest graphics to draw on the second dialog will be different styles of lines because you already have some experience drawing them. What you'll want to do is create one pen for each of the different pen styles, using the currently selected color. After you have created all of the pens, you'll loop through the different pens, selecting each one in turn and drawing a line across the dialog with each one. Before you start this loop, you need to perform a few calculations to determine where each of the lines should be on the dialog, with their starting and stopping points.

To begin adding this functionality to your application, you first add a color table, with one entry for each of the colors in the group of available colors on the first dialog. To create this color table, add a new member variable to the second dialog class, `CPaintDlg`, and specify the variable type as `static const COLORREF`, the name as `m_crColors[8]`, and the access as public. Open the source code file for the second dialog class, and add the color table in Listing 8.4 near the top of the file before the class constructor and destructor.

LISTING 8.4. THE COLOR TABLE.

```
1:  const COLORREF CPaintDlg::m_crColors[8] = {
2:      RGB(   0,   0,   0),    // Black
3:      RGB(   0,   0, 255),    // Blue
4:      RGB(   0, 255,   0),    // Green
5:      RGB(   0, 255, 255),    // Cyan
6:      RGB( 255,   0,   0),    // Red
7:      RGB( 255,   0, 255),    // Magenta
```

```
8:      RGB( 255, 255,   0),    // Yellow
9:      RGB( 255, 255, 255)     // White
10: };
11: ///////////////////////////////////////////////////////////////////////
12: // CPaintDlg dialog
.
.
.
```

With the color table in place, you can add a new function for drawing the lines. To keep the `OnPaint` function from getting too cluttered and difficult to understand, it makes more sense to place a limited amount of code in it to determine what should be drawn on the second dialog and then call other more specialized functions to draw the various shapes. With this in mind, you need to create a new member function for the second dialog class for drawing the lines. Declare this as a void function, and specify its declaration as `DrawLine(CPaintDC *pdc, int iColor)` and its access as private. You can edit this function, adding the code in Listing 8.5.

LISTING 8.5. THE `DrawLine` FUNCTION.

```
1:  void CPaintDlg::DrawLine(CPaintDC *pdc, int iColor)
2:  {
3:      // Declare and create the pens
4:      CPen lSolidPen (PS_SOLID, 1, m_crColors[iColor]);
5:      CPen lDotPen (PS_DOT, 1, m_crColors[iColor]);
6:      CPen lDashPen (PS_DASH, 1, m_crColors[iColor]);
7:      CPen lDashDotPen (PS_DASHDOT, 1, m_crColors[iColor]);
8:      CPen lDashDotDotPen (PS_DASHDOTDOT, 1, m_crColors[iColor]);
9:      CPen lNullPen (PS_NULL, 1, m_crColors[iColor]);
10:     CPen lInsidePen (PS_INSIDEFRAME, 1, m_crColors[iColor]);
11:
12:     // Get the drawing area
13:     CRect lRect;
14:     GetClientRect(lRect);
15:     lRect.NormalizeRect();
16:
17:     // Calculate the distance between each of the lines
18:     CPoint pStart;
19:     CPoint pEnd;
20:     int liDist = lRect.Height() / 8;
21:     CPen *lOldPen;
22:     // Specify the starting points
23:     pStart.y = lRect.top;
24:     pStart.x = lRect.left;
```

continues

LISTING 8.5. CONTINUED

```
25:     pEnd.y = pStart.y;
26:     pEnd.x = lRect.right;
27:     int i;
28:     // Loop through the different pens
29:     for (i = 0; i < 7; i++)
30:     {
31:         // Which pen are we on?
32:         switch (i)
33:         {
34:         case 0:    // Solid
35:             lOldPen = pdc->SelectObject(&lSolidPen);
36:             break;
37:         case 1:    // Dot
38:             pdc->SelectObject(&lDotPen);
39:             break;
40:         case 2:    // Dash
41:             pdc->SelectObject(&lDashPen);
42:             break;
43:         case 3:    // Dash Dot
44:             pdc->SelectObject(&lDashDotPen);
45:             break;
46:         case 4:    // Dash Dot Dot
47:             pdc->SelectObject(&lDashDotDotPen);
48:             break;
49:         case 5:    // Null
50:             pdc->SelectObject(&lNullPen);
51:             break;
52:         case 6:    // Inside
53:             pdc->SelectObject(&lInsidePen);
54:             break;
55:         }
56:         // Move down to the next position
57:         pStart.y = pStart.y + liDist;
58:         pEnd.y = pStart.y;
59:         // Draw the line
60:         pdc->MoveTo(pStart);
61:         pdc->LineTo(pEnd);
62:     }
63:     // Select the original pen
64:     pdc->SelectObject(lOldPen);
65: }
```

Now you need to edit the `OnPaint` function so that the `OnLine` function is called when it needs to be called. Add this function through the Class Wizard as an event-handler function for the `WM_PAINT` message. You'll notice that the generated code for this function creates a `CPaintDC` variable instead of the normal CDC class. The `CPaintDC` class is a

descendent of the CDC device context class. It automatically calls the `BeginPaint` and `EndPaint` API functions that all Windows applications must call before drawing any graphics during the `WM_PAINT` event message processing. It can be treated just like a regular device context object, calling all of the same functions.

When you are in the `OnPaint` function, you need to get a pointer to the parent window so that you can check the values of the variables tied to the groups of radio buttons to determine the color, tools, and shape to be drawn on the second dialog. This information tells you whether to call the `DrawLine` function or another function that you haven't written yet.

To add this functionality to your application, add an event handler for the `WM_PAINT` message on the second dialog class, adding the code in Listing 8.6 to the function created in your class.

LISTING 8.6. THE `OnPaint` FUNCTION.

```
1:  void CPaintDlg::OnPaint()
2:  {
3:      CPaintDC dc(this); // device context for painting
4:
5:      // TODO: Add your message handler code here
6:
7:      // Get a pointer to the parent window
8:      CGraphicsDlg *pWnd = (CGraphicsDlg*)GetParent();
9:      // Do we have a valid pointer?
10:     if (pWnd)
11:     {
12:         // Is the tool a bitmap?
13:         if (pWnd->m_iTool == 2)
14:         {
15:         }
16:         else    // No, we're drawing a shape
17:         {
18:             // Are we drawing a line?
19:             if (pWnd->m_iShape == 0)
20:                 DrawLine(&dc, pWnd->m_iColor);
21:         }
22:     }
23:     // Do not call CDialog::OnPaint() for painting messages
24:}
```

At this point, if you compile and run your application, you should be able to draw lines across the second dialog, as shown in Figure 8.4.

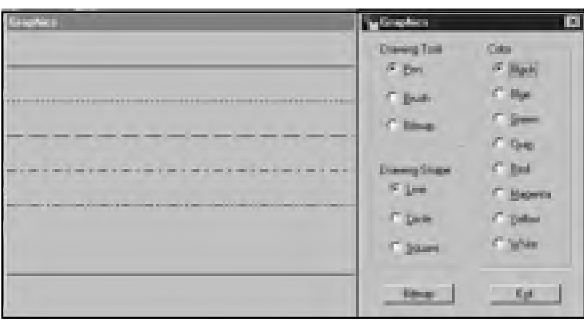


FIGURE 8.4.
Drawing lines on the second dialog.

Drawing Circles and Squares

Now that you have the basic structure in place, and you can see how you can change what is drawn on the second dialog at will, you are ready to add code to the second dialog to draw the circles and squares. To draw these figures, you use the `Ellipse` and `Rectangle` device context functions. These functions will use the currently selected pen and brush to draw these figures at the specified location. With both functions, you pass a `CRect` object to specify the rectangle in which to draw the specified figure. The `Rectangle` function fills the entire space specified, and the `Ellipse` function draws a circle or ellipse where the middle of each side of the rectangle touches the edge of the ellipse. Because these functions use both the pen and brush, you'll need to create and select an invisible pen and invisible brush to allow the user to choose either the pen or the brush. For the pen, you can use the null pen for this purpose, but for the brush, you'll need to create a solid brush the color of the window background (light gray).

When you calculate the position for each of these figures, you need to take a different approach from what you used with the lines. With the lines, you were able to get the height of the window, divide it by 8, and then draw a line at each of the divisions from the left edge to the right edge. With the ellipses and rectangles, you'll need to divide the dialog window into eight even rectangles. The easiest way to do this is to create two rows of figures with four figures in each row. Leave a little space between each figure so that the user can see the different pens used to outline each figure.

To add this functionality to your application, add a new function to the second dialog class. Specify the function type as void, the declaration as `DrawRegion(CPaintDC *pdc, int iColor, int iTool, int iShape)`, and the access as private. Edit the code in this function, adding the code in Listing 8.7.

LISTING 8.7. THE `DrawRegion` FUNCTION.

```
1:  void CPaintDlg::DrawRegion(CPaintDC *pdc, int iColor, int iTool, int
    ➥iShape)
2:  {
3:      // Declare and create the pens
4:      CPen lSolidPen (PS_SOLID, 1, m_crColors[iColor]);
5:      CPen lDotPen (PS_DOT, 1, m_crColors[iColor]);
6:      CPen lDashPen (PS_DASH, 1, m_crColors[iColor]);
7:      CPen lDashDotPen (PS_DASHDOT, 1, m_crColors[iColor]);
8:      CPen lDashDotDotPen (PS_DASHDOTDOT, 1, m_crColors[iColor]);
9:      CPen lNullPen (PS_NULL, 1, m_crColors[iColor]);
10:     CPen lInsidePen (PS_INSIDEFRAME, 1, m_crColors[iColor]);
11:
12:     // Declare and create the brushes
13:     CBrush lSolidBrush(m_crColors[iColor]);
14:     CBrush lBDiagBrush(HS_BDIAGONAL, m_crColors[iColor]);
15:     CBrush lCrossBrush(HS_CROSS, m_crColors[iColor]);
16:     CBrush lDiagCrossBrush(HS_DIAGCROSS, m_crColors[iColor]);
17:     CBrush lFDiagBrush(HS_FDIAGONAL, m_crColors[iColor]);
18:     CBrush lHorizBrush(HS_HORIZONTAL, m_crColors[iColor]);
19:     CBrush lVertBrush(HS_VERTICAL, m_crColors[iColor]);
20:     CBrush lNullBrush(RGB(192, 192, 192));
21:
22:     // Calculate the size of the drawing regions
23:     CRect lRect;
24:     GetClientRect(lRect);
25:     lRect.NormalizeRect();
26:     int liVert = lRect.Height() / 2;
27:     int liHeight = liVert - 10;
28:     int liHorz = lRect.Width() / 4;
29:     int liWidth = liHorz - 10;
30:     CRect lDrawRect;
31:     CPen *lOldPen;
32:     CBrush *lOldBrush;
33:     int i;
34:     // Loop through all of the brushes and pens
35:     for (i = 0; i < 7; i++)
36:     {
37:         switch (i)
38:         {
39:         case 0:    // Solid
40:             // Determine the location for this figure.
41:             // Start the first row
42:             lDrawRect.top = lRect.top + 5;
43:             lDrawRect.left = lRect.left + 5;
44:             lDrawRect.bottom = lDrawRect.top + liHeight;
45:             lDrawRect.right = lDrawRect.left + liWidth;
46:             // Select the appropriate pen and brush
```

continues

LISTING 8.7. CONTINUED

```
47:             lOldPen = pdc->SelectObject(&lSolidPen);
48:             lOldBrush = pdc->SelectObject(&lSolidBrush);
49:             break;
50:         case 1:    // Dot - Back Diagonal
51:             // Determine the location for this figure.
52:             lDrawRect.left = lDrawRect.left + liHorz;
53:             lDrawRect.right = lDrawRect.left + liWidth;
54:             // Select the appropriate pen and brush
55:             pdc->SelectObject(&lDotPen);
56:             pdc->SelectObject(&lBDiagBrush);
57:             break;
58:         case 2:    // Dash - Cross Brush
59:             // Determine the location for this figure.
60:             lDrawRect.left = lDrawRect.left + liHorz;
61:             lDrawRect.right = lDrawRect.left + liWidth;
62:             // Select the appropriate pen and brush
63:             pdc->SelectObject(&lDashPen);
64:             pdc->SelectObject(&lCrossBrush);
65:             break;
66:         case 3:    // Dash Dot - Diagonal Cross
67:             // Determine the location for this figure.
68:             lDrawRect.left = lDrawRect.left + liHorz;
69:             lDrawRect.right = lDrawRect.left + liWidth;
70:             // Select the appropriate pen and brush
71:             pdc->SelectObject(&lDashDotPen);
72:             pdc->SelectObject(&lDiagCrossBrush);
73:             break;
74:         case 4:    // Dash Dot Dot - Forward Diagonal
75:             // Determine the location for this figure.
76:             // Start the second row
77:             lDrawRect.top = lDrawRect.top + liVert;
78:             lDrawRect.left = lRect.left + 5;
79:             lDrawRect.bottom = lDrawRect.top + liHeight;
80:             lDrawRect.right = lDrawRect.left + liWidth;
81:             // Select the appropriate pen and brush
82:             pdc->SelectObject(&lDashDotDotPen);
83:             pdc->SelectObject(&lFDiagBrush);
84:             break;
85:         case 5:    // Null - Horizontal
86:             // Determine the location for this figure.
87:             lDrawRect.left = lDrawRect.left + liHorz;
88:             lDrawRect.right = lDrawRect.left + liWidth;
89:             // Select the appropriate pen and brush
90:             pdc->SelectObject(&lNullPen);
91:             pdc->SelectObject(&lHorizBrush);
92:             break;
93:         case 6:    // Inside - Vertical
94:             // Determine the location for this figure.
```

```
95:             lDrawRect.left = lDrawRect.left + liHorz;
96:             lDrawRect.right = lDrawRect.left + liWidth;
97:             // Select the appropriate pen and brush
98:             pdc->SelectObject(&lInsidePen);
99:             pdc->SelectObject(&lVertBrush);
100:            break;
101:        }
102:        // Which tool are we using?
103:        if (iTool == 0)
104:            pdc->SelectObject(lNullBrush);
105:        else
106:            pdc->SelectObject(lNullPen);
107:        // Which shape are we drawing?
108:        if (iShape == 1)
109:            pdc->Ellipse(lDrawRect);
110:        else
111:            pdc->Rectangle(lDrawRect);
112:    }
113:    // Reset the original brush and pen
114:    pdc->SelectObject(lOldBrush);
115:    pdc->SelectObject(lOldPen);
116:}
```

Now that you have the capability to draw the circles and squares in the second dialog, you'll need to call this function when the user has selected either of these two figures with either a pen or a brush. To do this, add the two lines starting at line 21 in Listing 8.8 to the `OnPaint` function.

LISTING 8.8. THE MODIFIED `OnPaint` FUNCTION.

```
1:  void CPaintDlg::OnPaint()
2:  {
3:      CPaintDC dc(this); // device context for painting
4:
5:      // TODO: Add your message handler code here
6:
7:      // Get a pointer to the parent window
8:      CGraphicsDlg *pWnd = (CGraphicsDlg*)GetParent();
9:      // Do we have a valid pointer?
10:     if (pWnd)
11:     {
12:         // Is the tool a bitmap?
13:         if (pWnd->m_iTool == 2)
14:         {
15:         }
16:         else    // No, we're drawing a shape
```

continues

LISTING 8.8. CONTINUED

```
17:         {
18:             // Are we drawing a line?
19:             if (m_iShape == 0)
20:                 DrawLine(&dc, pWnd->m_iColor);
21:             else    // We're drawing a ellipse or rectangle
22:                 DrawRegion(&dc, pWnd->m_iColor, pWnd->m_iTool,
    ➥pWnd->m_iShape);
23:         }
24:     }
25:     // Do not call CDialog::OnPaint() for painting messages
26:}
```

Now you should be able to compile and run your application and display not only lines, but also squares and circles, switching between displaying the outlines and the filled-in figure without any outline, as shown in Figure 8.5.

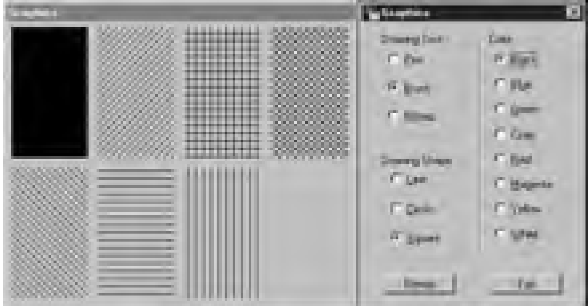


FIGURE 8.5.
Drawing rectangles on the second dialog.

Loading Bitmaps

Now that you can draw various graphic images on the second dialog window, all that's left is to add the functionality to load and display bitmaps. You could easily add the bitmaps to the resources in the application, give them their own object IDs, and then use the `LoadBitmap` and `MAKEINTRESOURCE` functions to load the bitmap into a `CBitmap` class object, but that isn't extremely useful when you start building your own applications. What would be really useful is the ability to load bitmaps from files on the computer disk. To provide this functionality, you use the `LoadImage` API function to load the bitmap images into memory and then attach the loaded image to the `CBitmap` object.

To do this in your application, you can attach a function to the bitmap button on the first dialog that displays the File Open dialog to the user, allowing the user to select a bitmap to be displayed. You'll want to build a filter for the dialog, limiting the available files to

bitmaps that can be displayed in the second dialog. After the user selects a bitmap, you'll get the file and path name from the dialog and load the bitmap using the `LoadImage` function. When you have a valid handle to the bitmap that was loaded into memory, you'll delete the current bitmap image from the `CBitmap` object. If there was a bitmap loaded into the `CBitmap` object, you'll detach the `CBitmap` object from the now deleted image. After you make sure that there isn't already an image loaded in the `CBitmap` object, you attach the image you just loaded into memory, using the `Attach` function. At this point, you want to invalidate the second dialog so that if it's displaying a bitmap, it displays the newly loaded bitmap.

To support this functionality, you need to add a string variable to hold the bitmap name, and a `CBitmap` variable to hold the bitmap image, to the first dialog class. Add these two variables as listed in Table 8.5.

TABLE 8.5. BITMAP VARIABLES.

Name	*Type*	*Access*
`m_sBitmap`	`CString`	`Public`
`m_bmpBitmap`	`CBitmap`	`Public`

After you add the variables to the first dialog class, add an event-handler function to the clicked event of the `Bitmap` button using the Class Wizard. After you add this function, edit it, adding the code in Listing 8.9.

LISTING 8.9. THE `OnBbitmap` FUNCTION.

```
1:  void CGraphicsDlg::OnBbitmap()
2:  {
3:      // TODO: Add your control notification handler code here
4:
5:      // Build a filter for use in the File Open dialog
6:      static char BASED_CODE szFilter[] = "Bitmap Files (*.bmp)¦*.bmp¦¦";
7:      // Create the File Open dialog
8:      CFileDialog m_ldFile(TRUE, ".bmp", m_sBitmap,
9:          OFN_HIDEREADONLY ¦ OFN_OVERWRITEPROMPT, szFilter);
10:
11:     // Show the File Open dialog and capture the result
12:     if (m_ldFile.DoModal() == IDOK)
13:     {
14:         // Get the filename selected
15:         m_sBitmap = m_ldFile.GetPathName();
16:         // Load the selected bitmap file
```

continues

LISTING 8.9. CONTINUED

```
17:         HBITMAP hBitmap = (HBITMAP) ::LoadImage(AfxGetInstanceHandle(),
18:             m_sBitmap, IMAGE_BITMAP, 0, 0,
19:             LR_LOADFROMFILE ¦ LR_CREATEDIBSECTION);
20:
21:         // Do we have a valid handle for the loaded image?
22:         if (hBitmap)
23:         {
24:             // Delete the current bitmap
25:             if (m_bmpBitmap.DeleteObject())
26:                 // If there was a bitmap, detach it
27:                 m_bmpBitmap.Detach();
28:             // Attach the currently loaded bitmap to the bitmap object
29:             m_bmpBitmap.Attach(hBitmap);
30:         }
31:         // Invalidate the second dialog window
32:         m_dlgPaint.Invalidate();
33:     }
34: }
```

Displaying Bitmaps

Now that you can load bitmaps into memory, you need to display them for the user. You need to copy the bitmap from the `CBitmap` object to a `BITMAP` structure, using the `GetBitmap` function, which will get the width and height of the bitmap image. Next, you'll create a new device context that is compatible with the screen device context. You'll select the bitmap into the new device context and then copy it from this second device context to the original device context, resizing it as it's copied, using the `StretchBlt` function.

To add this functionality to your application, add a new member function to the second dialog class. Specify the function type as void, the function declaration as `ShowBitmap(CPaintDC *pdc, CWnd *pWnd)`, and the function access as private. Edit the function, adding the code in Listing 8.10.

Notice that you have declared the window pointer being passed in as a pointer to a `CWnd` object, instead of the class type of your main dialog. To declare it as a pointer to the class type of the first dialog, you'd need to declare the class for the first dialog before the class declaration for the second dialog. Meanwhile, the first dialog requires that the second dialog class be declared first. This affects the order in which the include files are added to the source code at the top of each file. You cannot have both classes

declared before the other; one has to be first. Although there are ways to get around this problem, by declaring a place holder for the second class before the declaration of the first class, it's easier to cast the pointer as a pointer to the first dialog class in the function in this instance. To learn how to declare a place holder for the second class, see Appendix A, "C++ Review."

LISTING 8.10. THE `ShowBitmap` FUNCTION.

```
1:  void CPaintDlg::ShowBitmap(CPaintDC *pdc, CWnd *pWnd)
2:  {
3:      // Convert the pointer to a pointer to the main dialog class
4:      CGraphicsDlg *lpWnd = (CGraphicsDlg*)pWnd;
5:      BITMAP bm;
6:      // Get the loaded bitmap
7:      lpWnd->m_bmpBitmap.GetBitmap(&bm);
8:      CDC dcMem;
9:      // Create a device context to load the bitmap into
10:     dcMem.CreateCompatibleDC(pdc);
11:     // Select the bitmap into the compatible device context
12:     CBitmap* pOldBitmap = (CBitmap*)dcMem.SelectObject
        ➥(lpWnd->m_bmpBitmap);
13:     CRect lRect;
14:     // Get the display area available
15:     GetClientRect(lRect);
16:     lRect.NormalizeRect();
17:     // Copy and resize the bitmap to the dialog window
18:     pdc->StretchBlt(10, 10, (lRect.Width() - 20),
19:             (lRect.Height() - 20), &dcMem, 0, 0,
20:             bm.bmWidth, bm.bmHeight, SRCCOPY);
21: }
```

Now that you have the ability to display the currently selected bitmap on the dialog, you'll need to add the functionality to call this function to the `OnPaint` function in the second dialog. You can determine whether a bitmap has been specified by checking the value of the `m_sBitmap` variable on the first dialog. If this string is empty, there is no bitmap to be displayed. If the string is not empty, you can call the `ShowBitmap` function. To add this last bit of functionality to this application, edit the `OnPaint` function, adding lines 15 through 18 from Listing 8.11.

LISTING 8.11. THE MODIFIED `OnPaint` FUNCTION.

```
1:  void CPaintDlg::OnPaint()
2:  {
3:      CPaintDC dc(this); // device context for painting
4:
5:      // TODO: Add your message handler code here
6:
7:      // Get a pointer to the parent window
8:      CGraphicsDlg *pWnd = (CGraphicsDlg*)GetParent();
9:      // Do we have a valid pointer?
10:     if (pWnd)
11:     {
12:         // Is the tool a bitmap?
13:         if (pWnd->m_iTool == 2)
14:         {
15:             // Is there a bitmap selected and loaded?
16:             if (pWnd->m_sBitmap != "")
17:                 // Display it
18:                 ShowBitmap(&dc, pWnd);
19:         }
20:         else    // No, we're drawing a shape
21:         {
22:             // Are we drawing a line?
23:             if (m_iShape == 0)
24:                 DrawLine(&dc, pWnd->m_iColor);
25:             else    // We're drawing a ellipse or rectangle
26:                 DrawRegion(&dc, pWnd->m_iColor, pWnd->m_iTool,
27:                                 pWnd->m_iShape);
28:         }
29:     }
30:     // Do not call CDialog::OnPaint() for painting messages
31:}
```

At this point, you should be able to select a bitmap from your system and display it in the second dialog, as shown in Figure 8.6.

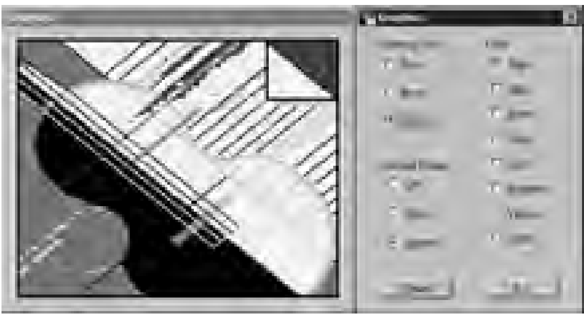


FIGURE 8.6.
Showing a bitmap in the second dialog.

Summary

What a way to start the week! You learned a lot today. You learned how Windows uses device context objects to allow you to draw graphics in the same way every time, without having to worry about what hardware users might have in their computers. You learned about some of the basic GDI objects, such as pens and brushes, and how they are used to draw figures on windows and dialogs. You also learned how you can load bitmaps from the system disk and display them onscreen for the user to see. You learned about the different pen and brush styles and how you can use these to draw the type of figure you want to draw. You also learned how you can specify colors for use with pens and brushes so that you can control how images appear to the user.

Q&A

Q Why do I need to specify both a pen and a brush if I just want to display one or the other?

A You are always drawing with both when you draw any object that is filled in. The pen draws the outline, and the brush fills in the interior. You cannot choose to use one or the other; you have to use both. If you only want to display one or the other, you need to take special steps.

Q Why do all of the pen styles become solid when I increase the pen width above 1?

A When you increase the pen width, you are increasing the size of the dot that is used to draw with. If you remember Day 3, "Allowing User Interaction—Integrating the Mouse and Keyboard in Your Application," when you first tried to draw by capturing each spot the mouse covered, all you drew were a bunch of dots. Well, when you increase the size of the dots that you are drawing the line with, the gaps between the dots are filled in from both sides, providing an unbroken line.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What are the three values that are combined to specify a color?
2. What do you use to draw on windows without needing to know what graphics card the user has?
3. What size bitmap can you use to make a brush from it?
4. What event message is sent to a window to tell it to redraw itself?
5. How can you cause a window to repaint itself?

Exercises

1. Make the second dialog window resizable, and make it adjust the figures drawn on it whenever it's resized.
2. Add a bitmap brush to the set of brushes used to create the rectangles and ellipses.

WEEK 2

DAY 9

Adding ActiveX Controls to Your Application

In today's application develop market, there are thousands of prebuilt components that you can plug into your applications, extending the functionality of your applications instantaneously. Originally the domain of Visual Basic programmers, now you can use readily available ActiveX controls with just about any Windows programming language, including Visual C++. Today you will learn how you can add ActiveX controls to your Visual C++ applications, taking advantage of their existing functionality. Some of the topics that you will cover today are

- What ActiveX controls are and how they work.
- How you can add ActiveX controls to your project workspace.
- How you can use the ActiveX control in your Visual C++ application.
- How to call the various methods associated with the ActiveX control.
- How to handle events that are triggered by the ActiveX control.

What Is an ActiveX Control?

An ActiveX control is a software component that can be plugged into many different programs and used as if it were a native part of the program. It's similar to the concept of separate stereo components. If you buy a new tape deck, you can just plug it into the rest of your stereo and it works with everything else you already have. ActiveX controls bring this same type of interoperability to software applications.

ActiveX used to be called OLE 2.0. OLE 2.0 was Microsoft's technology for combining two or more applications to make them work as one (or at least to switch between the various applications within the same application shell). This idea was an expansion from the original OLE (Object Linking and Embedding) technology, which only enabled you to combine documents created with different applications into a single document. When revamping OLE technologies to work in a distributed environment (such as on the Internet), Microsoft decided to also revamp the name. Thus, ActiveX was born.

ActiveX and the `IDispatch` Interface

The ActiveX technology is built on top of Microsoft's COM (Component Object Model) technology, utilizing its interface and interaction model for making ActiveX control integration fairly seamless. The COM technology defines how ActiveX objects are constructed and how their interfaces are designed. The ActiveX technology defines a layer that is built on top of COM, what interfaces various objects should support, and how different types of objects should interact.

Microsoft's COM technology defines how applications and components can interact through the use of interfaces. An interface is like a function call into an ActiveX component. However, COM specifies how that function call must be built and called, and what supporting functionality must accompany the function call.

There are interfaces, like the `IUnknown` interface, that are required in every COM object, and which are used to query the component to find out what other interfaces are supported by the component. Each interface supports a specific set of functionality; you might have one interface to handle the visual appearance of the control, another to control how the control appearance interacts with the surrounding application, another that triggers events in the surrounding application, and so on.

One of the key technologies in ActiveX controls is *automation*. Automation enables an application embedded within another application to activate itself and control its part of the user interface or document, making its changes and then shutting itself down when the user moves on to another part of the application that isn't controlled by the embedded application.

This process is what happens when you have an Excel spreadsheet embedded within a Word document. If you click the spreadsheet, Excel becomes active and you can edit the spreadsheet using Excel, even though you're still working in Word. Then, once you finish making your changes to the spreadsheet, Excel closes itself down and you can continue working in Word.

One of the keys to making automation work is a special interface called the `IDispatch` (also known as the dispinterface) interface. The `IDispatch` interface consists of a pointer to a table of available methods that can be run in the ActiveX control or embedded application. These methods have ID numbers, called DISPIDs, which are also loaded into a table that can be used to look up the ID for a specific method. Once you know the DISPID for a specific method, you can call that method by calling the `Invoke` method of the `IDispatch` interface, passing the DISPID to identify the method to be run. Figure 9.1 shows how the `IDispatch` interface uses the `Invoke` method to run methods in the ActiveX object.

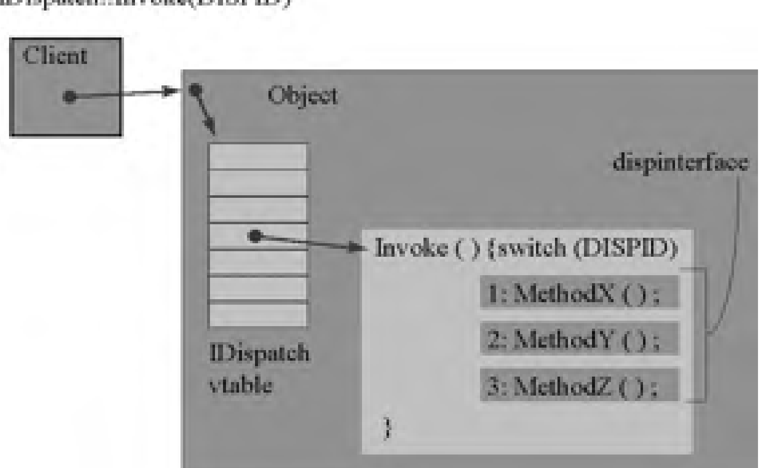


FIGURE 9.1.
The `IDispatch` *ActiveX interface.*

ActiveX Containers and Servers

To embed one ActiveX object within another ActiveX object, you have to implement the embedded object as an ActiveX *server*, and the object containing the first object must be an ActiveX *container*. Any ActiveX object that can be embedded within another is an ActiveX server, whether it is an entire application or just a small ActiveX control. Any ActiveX object that can have other ActiveX objects embedded within it is an ActiveX container.

Don't confuse the use of the terms *container* and *server* with the term *client* in the previous figure. The client is the object calling the other object's `IDispatch` interface. As you'll learn in a page or so, both the container and server call the other's `IDispatch` interfaces, making each one the client of the other.

These two types of ActiveX objects are not mutually exclusive. An ActiveX server can also be an ActiveX container. A good example of this concept is Microsoft's Internet Explorer Web browser. Internet Explorer is implemented as an ActiveX server that runs within an ActiveX container shell (that can also house Word, Excel, PowerPoint, or any other ActiveX server application). At the same time that Internet Explorer is an ActiveX server running within the browser shell, it can contain other ActiveX controls.

ActiveX controls are a special instance of an ActiveX server. Some ActiveX servers are also applications that can run on their own. ActiveX controls cannot run on their own and must be embedded within an ActiveX container. By using ActiveX components in your Visual C++ application, you automatically make your application an ActiveX container.

Most of the interaction between the ActiveX container and an ActiveX control takes place through three `IDispatch` interfaces. One of these `IDispatch` interfaces is on the control, and it is used by the container to make calls to the various methods that the ActiveX control makes available to the container.

The container provides two `IDispatch` interfaces to the control. The first of these `IDispatch` interfaces is used by the control to trigger events in the container application. The second interface is used to set properties of the control, as shown in Figure 9.2. Most properties of an ActiveX control are actually provided by the container but are maintained by the control. When you set a property for the control, the container calls a method in the control to tell the control to read the properties from the container. Most of this activity is transparent to you because Visual C++ builds a series of C++ classes around the ActiveX control's interfaces. You will interact with the methods exposed by the C++ classes, not directly calling the control's `IDispatch` interface.

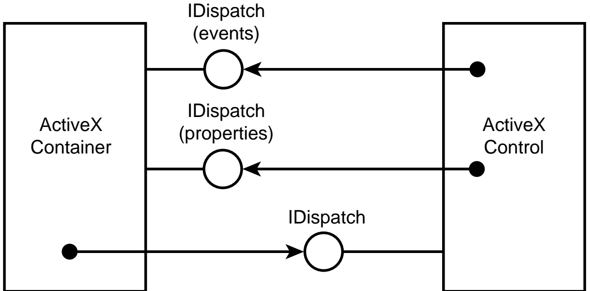


FIGURE 9.2.
An ActiveX container and control interact primarily through a few `IDispatch` *interfaces.*

Adding an ActiveX Control to Your Project

Looking into how ActiveX controls work can be deceptive because of how easy it really is to use them in your applications. Visual C++ makes it easy to add ActiveX controls to your applications and even easier to use them. Before you begin adding the ActiveX control to your application, let's create an application shell into which you will add an ActiveX control:

1. Create a new MFC AppWizard project named `ActiveX`.
2. Use the same defaults on the AppWizard as in previous days, but leave the check box for ActiveX Controls checked on the second AppWizard step. Give your application the title `ActiveX Controls`.
3. Once you generate an application shell, remove all the controls and add a single command button.
4. Set the button's ID to `IDC_EXIT` and its caption to `E&xit`.
5. Using the Class Wizard, add a function to your command button on the `BN_CLICKED` event message.
6. Edit the function you just created, calling the `OnOK` function, as on earlier days.

Registering the Control

Before you add an ActiveX control to your dialog window, you need to register the control, both with Windows and with Visual C++. There are two possible ways to register the ActiveX control with Windows. The first way is to run any installation routine that came with the ActiveX control. If you do not have an installation routine, you need to register the control manually. To register the control manually, follow these steps:

1. Open a DOS shell.
2. Change directory to where the ActiveX control is on your system.
3. Run the `regsvr32` command, specifying the name of the ActiveX control as the

only command-line argument. For instance, if you were registering a control named `MYCTL.OCX` and it was located in your `WINDOWS\SYSTEM` directory, you would perform the following:

```
C:\WINDOWS> CD system
C:\WINDOWS\SYSTEM> regsvr32 MYCTL.OCX
```

It is preferable to run any installation routine that comes with the control because registering the control manually might not enable the control for development usage. Controls can be licensed for development or deployment. If a control is licensed for deployment, you will not be able to use it in your Visual C++ applications. This is a mechanism that protects control developers by requiring that developers purchase a development license for controls; they can't just use the controls they may have installed on their system with another application.

COM and ActiveX objects store a lot of information in the Windows Registry database. Whenever an application uses an ActiveX object, the operating system refers to the information in the Windows Registry to find the object and to determine whether the application can use the object in the way that it requested. Using the `regsvr32.exe` utility to register an ActiveX control places most of the required information about the control into the system Registry. However, there may be additional information about the control that needs to be in the Registry for the control to function properly.

Now that the ActiveX control that you want to use is registered with the operating system, you need to register it with Visual C++ and add it to your project. To do this, follow these steps:

1. Select Project | Add To Project | Components and Controls from the Visual C++ menu.
2. In the Components and Controls Gallery dialog, navigate to the Registered ActiveX Controls folder, as in Figure 9.3.

FIGURE 9.3.

The ActiveX controls that can be added to your project.

3. Select the control you want to register, such as the Microsoft FlexGrid control, and click the Insert button.
4. Click OK on the message box asking whether you want to insert this component in your project.
5. On the Confirm Classes dialog, click the OK button to add the C++ classes specified, as in Figure 9.4.

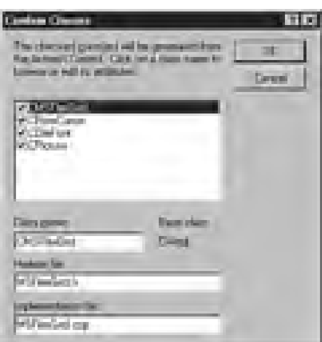


FIGURE 9.4.

Visual C++ tells you what classes will be added to your project.

6. Click the Close button on the Components and Controls Gallery dialog to finish adding controls to your project.
7. The FlexGrid control should have been added to the Control Palette for your dialog window, as in Figure 9.5.

FIGURE 9.5.

The ActiveX control FlexGrid is added to the Control Palette for use on your dialog windows.

If you examine the Class View area of the workspace pane, you see the four classes that Visual C++ added to your project. Expand the class trees and you see numerous methods for these classes. Visual C++ created these classes and methods by examining the ActiveX control that you just added and created class methods to call each of the methods in the control's `IDispatch` interface.

If you use older ActiveX controls in your Visual C++ applications, Visual C++ might not be able to generate the classes and methods to encapsulate the control's functionality. The information in the control that provided Visual C++ with the information necessary to build these classes and methods is a more recent addition to the ActiveX specification. As a result, older controls might not provide this information, making them more difficult to use with Visual C++.

Adding the Control to Your Dialog

Now that you have added the FlexGrid control to your project, you can add it to your dialog window just as you would any other control. Set the control properties as in Table 9.1.

TABLE 9.1. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
FlexGrid control	ID	`IDC_MSFGRID`
	Rows	`20`
	Cols	`4`
	MergeCells	`2 - Restrict Rows`
	Format (FormatString)	`< Region ¦< Product ¦< Employee ¦>Sales`

Once you add the control to your dialog window, you will notice that there is an additional tab on the properties dialog with all the control properties, as in Figure 9.6. You can choose to use this tab to set all the properties on the control, or you can go through the other tabs to set the properties, just as you would with the standard controls.

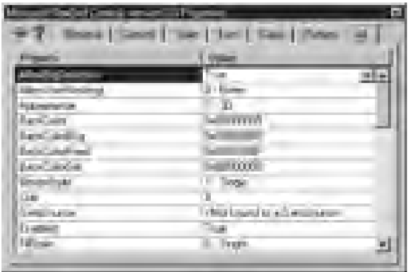


FIGURE 9.6.
ActiveX controls have a property tab that contains all control properties.

Once you have finished setting all the properties for the control, you'll need to add a variable for the control so that you can interact with the control in your code. To add this variable, open the Member Variables tab on the Class Wizard and add a variable for the control. Because you are adding a variable for an ActiveX control, you can only add a control variable, so the only thing available for you to specify is the variable name. For this example application, name the variable `m_ctlFGrid`.

Using an ActiveX Control in Your Application

Once Visual C++ has generated all the classes to encapsulate the ActiveX control, working with the control is a simple matter of calling the various methods and responding to control events just like the standard controls. You'll start with using the control methods to get information about the control and to modify data within the control. Then you'll learn how to respond to control events with Visual C++.

Interacting with the Control

The application that you are building today will generate a number of product sales over five sales regions with four salespeople. You will be able to scroll through the data, which will be sorted by region and product, to compare how each salesperson did for each product.

To make this project, you will build an array of values that will be loaded into cells in the grid. The grid will then be sorted in ascending order, using the FlexGrid control's internal sorting capabilities.

Loading Data into the Control

The first thing you will do is create a function to load data into the FlexGrid control. Add a new function to the `CActiveXDlg` class by right-clicking the Class View of the workspace and choosing Add Member Function. Specify the Function Type as `void`, the Function Declaration as `LoadData`, and the access as Private. Click the OK button and edit the function, adding the code in Listing 9.1.

LISTING 9.1. THE `LoadData` FUNCTION.

```
 1: void CActiveXDlg::LoadData()
 2: {
 3:     int liCount;        // The grid row count
 4:     CString lsAmount;    // The sales amount
 5:
 6:     // Initialize the random number generator
 7:     srand((unsigned)time(NULL));
 8:     // Create Array in the control
 9:     for (liCount = m_ctlFGrid.GetFixedRows();
10:         liCount < m_ctlFGrid.GetRows(); liCount++)
11:     {
12:         // Generate the first column (region) values
13:         m_ctlFGrid.SetTextArray(GenID(liCount, 0), RandomStringValue(0));
14:         // Generate the second column (product) values
15:         m_ctlFGrid.SetTextArray(GenID(liCount, 1), RandomStringValue(1));
16:         // Generate the third column (employee) values
17:         m_ctlFGrid.SetTextArray(GenID(liCount, 2), RandomStringValue(2));
18:         // Generate the sales amount values
19:         lsAmount.Format("%5d.00", rand());
20:         // Populate the fourth column
21:         m_ctlFGrid.SetTextArray(GenID(liCount, 3), lsAmount);
22:     }
23:
24:     // Merge the common subsequent rows in these columns
25:     m_ctlFGrid.SetMergeCol(0, TRUE);
26:     m_ctlFGrid.SetMergeCol(1, TRUE);
27:     m_ctlFGrid.SetMergeCol(2, TRUE);
28:
29:     // Sort the grid
30:     DoSort();
31: }
```

In this function, the first thing that you do is initialize the random number generator. Next, you loop through all of the rows in the control, placing data in each of the cells. You get the total number of rows in the control by calling the `GetRows` method and the number of the header row by calling the `GetFixedRows` method. You are able to add data

to the control cells by calling the `SetTextArray` method, which has the cell ID as the first argument and the cell contents as the second argument, both of which are generated by functions you'll be creating in a few moments.

Once you have data in the grid cells, you call `SetMergeCol`, which tells the control that it can merge cells in the first three columns if adjacent rows contain the same value. Finally, you sort the control, using another function you have yet to create.

Calculating the Cell ID

The cells in the FlexGrid control are numbered sequentially from left to right, top to bottom. With your control, the first row, which contains the headers (and is already populated), has cells 0 through 3, the second row cells 4 through 7, and so on. Therefore, you can calculate the ID of a cell by adding its column number to the total number of columns in the control, multiplied by the current row number. For instance, if your control has four columns, and you are in the third column and fourth row, you can calculate your cell ID as 2 + (4 * 3) = 14. (Remember that the column and row numbers start with 0, so the third column is 2 and the fourth row is number 3.)

Now that you understand how you can calculate the cell ID, you need to implement that formula in a function. Add a new function to the `CActiveXDlg` class using the same method as for the `LoadData` function. The type of this function should be `int` and the description should be `GenID(int m_iRow, int m_iCol)`. Once you add the function, edit it with the code in Listing 9.2.

LISTING 9.2. THE `GenID` FUNCTION.

```
1: int CActiveXDlg::GenID(int m_iRow, int m_iCol)
2: {
3:     // Get the number of columns
4:     int liCols = m_ctlFGrid.GetCols();
5:
6:     // Generate an ID based on the number of columns,
7:     // the current column, and the current row
8:     return (m_iCol + liCols * m_iRow);
9: }
```

Generating Random Data

To populate the first three columns in the grid, you want to randomly generate data. In the first column, you want to put region names. In the second column, you want to put product names. And in the third column, you want to put salesperson names. By using a switch statement to determine which column you are generating data for and then using a

modulus division on a randomly generated number in another switch statement, you can randomly select between a limited set of data strings.

To implement this functionality, add another function to the `CActiveXDlg` class with a type of `CString` and a description of `RandomStringValue(int m_iColumn)`. Edit the resulting function, adding the code in Listing 9.3.

LISTING 9.3. THE `RandomStringValue` FUNCTION.

```
 1: CString CActiveXDlg::RandomStringValue(int m_iColumn)
 2: {
 3:     CString lsStr;    // The return string
 4:     int liCase;        // A random value ID
 5:
 6:     // Which column are we generating for?
 7:     switch (m_iColumn)
 8:     {
 9:     case 0:    // The first column (region)
10:         // Generate a random value between 0 and 4
11:         liCase = (rand() % 5);
12:         // What value was generated?
13:         switch (liCase)
14:         {
15:         case 0:
16:             // 0 - Northwest region
17:             lsStr = "Northwest";
18:             break;
19:         case 1:
20:             // 1 - Southwest region
21:             lsStr = "Southwest";
22:             break;
23:         case 2:
24:             // 2 - Midwest region
25:             lsStr = "Midwest";
26:             break;
27:         case 3:
28:             // 3 - Northeast region
29:             lsStr = "Northeast";
30:             break;
31:         default:
32:             // 4 - Southeast region
33:             lsStr = "Southeast";
34:             break;
35:         }
36:         break;
37:     case 1:    // The second column (product)
38:         // Generate a random value between 0 and 4
39:         liCase = (rand() % 5);
40:         // What value was generated?
41:         switch (liCase)
```

```
42:         {
43:         case 0:
44:             // 0 - Dodads
45:             lsStr = "Dodads";
46:             break;
47:         case 1:
48:             // 1 - Thingamajigs
49:             lsStr = "Thingamajigs";
50:             break;
51:         case 2:
52:             // 2 - Whatchamacallits
53:             lsStr = "Whatchamacallits";
54:             break;
55:         case 3:
56:             // 3 - Round Tuits
57:             lsStr = "Round Tuits";
58:             break;
59:         default:
60:             // 4 - Widgets
61:             lsStr = "Widgets";
62:             break;
63:         }
64:         break;
65:     case 2:    // The third column (employee)
66:         // Generate a random value between 0 and 3
67:         liCase = (rand() % 4);
68:         // What value was generated?
69:         switch (liCase)
70:         {
71:         case 0:
72:             // 0 - Dore
73:             lsStr = "Dore";
74:             break;
75:         case 1:
76:             // 1 - Harvey
77:             lsStr = "Harvey";
78:             break;
79:         case 2:
80:             // 2 - Pogo
81:             lsStr = "Pogo";
82:             break;
83:         default:
84:             // 3 - Nyra
85:             lsStr = "Nyra";
86:             break;
87:         }
88:         break;
89:     }
90:     // Return the generated string
91:     return lsStr;
92: }
```

Sorting the Control

To sort the Grid control, you need to select all the columns and then set the sort to ascending. To implement this functionality, add one more function to the `CActiveXDlg` class with a type of `void` and a definition of `DoSort`. Edit the function as in Listing 9.4.

LISTING 9.4. THE `DoSort` FUNCTION.

```
1: void CActiveXDlg::DoSort()
2: {
3:     // Set the current column to column 0
4:     m_ctlFGrid.SetCol(0);
5:     // Set the column selection to all columns
6:     m_ctlFGrid.SetColSel((m_ctlFGrid.GetCols() - 1));
7:     // Generic Ascending Sort
8:     m_ctlFGrid.SetSort(1);
9: }
```

In the `DoSort` function, you set the current column to the first column using the `SetCol` method. Next you select from the current column to the last column using the `SetColSel` method, effectively selecting all columns in the control. Finally, you tell the control to sort the columns in ascending order by using the `SetSort` method, passing `1` as the flag for the sort order.

Now that you have all the functionality necessary to load the control with data, you need to call the `LoadData` function in the `OnInitDialog` function to load the data before the control is visible to the user. Edit the `OnInitDialog` function as in Listing 9.5 to load the data.

LISTING 9.5. THE `OnInitDialog` FUNCTION.

```
 1: BOOL CActiveXDlg::OnInitDialog()
 2: {
 3:     CDialog::OnInitDialog();
 4: .
 5: .
 6: .
 7:     // TODO: Add extra initialization here
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Load data into the Grid control
14:     LoadData();
```

```
15:
16:     ///////////////////////
17:     // MY CODE ENDS HERE
18:     ///////////////////////
19:
20:     return TRUE;  // return TRUE  unless you set the focus to a control
21: }
```


If you compile and run your application at this point, you find that it is loading the data and sorting it, as in Figure 9.7.

FIGURE 9.7.
The FlexGrid populated with data.

Responding to Control Events

If you play with your application at this point, you know that the Grid control does not respond to any input that you might try to give it. If you click one of the cells and try to change the value, it doesn't respond. What you need to do is add a control event to handle the input. ActiveX controls make several events available for use in Visual C++ applications. You can use the Class Wizard to browse through the available events and determine which events you need to give functionality and which to ignore. Most ActiveX controls don't have any default functionality attached to the available events but instead expect you to tell the control what to do on each event.

You are going to add two control events to capture the mouse clicks and movements. You will add functionality to allow the user to click a column header and drag it to another position, thus rearranging the column order. To implement this functionality, you have to capture two control events, when the mouse button is pressed down and when it is released. On the first event, you need to check whether the user clicked a header, and if so, you capture the column selected. On the second event, you need to move the selected column to the column on which the mouse button was released.

To accomplish this functionality, you need to create a new class variable to maintain the clicked column number between the two events. Add a new variable to the `CActiveXDlg` class, just like you added the functions earlier, specifying the type as `int`, the variable name as `m_iMouseCol`, and the access as Private.

Capturing the Column Selected

To capture the mouse click event for the control, follow these steps:

1. Using the Class Wizard, add a function for the `MouseDown` event message for the `IDC_MSFGRID` object.
2. Edit the function using the code in Listing 9.6.

LISTING 9.6. THE `OnMouseDownMsfgrid` FUNCTION.

```
 1: void CActiveXDlg::OnMouseDownMsfgrid(short Button, short Shift, long
    ➥x, long y)
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Did the user click on a data row and not the
10:     // header row?
11:     if (m_ctlFGrid.GetMouseRow() != 0)
12:     {
13:         // If so, then zero out the column variable
14:         // and exit
15:         m_iMouseCol = 0;
16:         return;
17:     }
18:     // Save the column clicked on
19:     m_iMouseCol = m_ctlFGrid.GetMouseCol();
20:
21:     ///////////////////////
22:     // MY CODE ENDS HERE
23:     ///////////////////////
24: }
```

In this function, you checked the row clicked by calling the `GetMouseRow` method. If the row is not the first row, then zero out the column-holding variable and exit the function. Otherwise, you need to get the column clicked by calling the `GetMouseCol` method. You can store the returned column number in the `m_iMouseCol` variable that you just added to the class.

Moving the Column Where Released

Now that you are capturing the selected column number, you need to capture the column on which the mouse is released. To capture the mouse release event for the control, follow these steps:

1. Using the Class Wizard, add a function for the `MouseUp` event message for the `IDC_MSFGRID` object.
2. Edit the function using the code in Listing 9.7.

LISTING 9.7. THE `OnMouseUpMsfgrid` FUNCTION.

```
 1: void CActiveXDlg::OnMouseUpMsfgrid(short Button, short Shift, long x,
    ➥long y)
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // If the selected column was the first column,
10:     // there's nothing to do
11:     if (m_iMouseCol == 0)
12:         return;
13:     // Turn the control redraw off
14:     m_ctlFGrid.SetRedraw(FALSE);
15:     // Change the selected column position
16:     m_ctlFGrid.SetColPosition(m_iMouseCol, m_ctlFGrid.GetMouseCol());
17:     // Resort the grid
18:     DoSort();
19:     // Turn redraw back on
20:     m_ctlFGrid.SetRedraw(TRUE);
21:
22:     ///////////////////////
23:     // MY CODE ENDS HERE
24:     ///////////////////////
25: }
```

In this function, you first check to see if there is a selected column to be moved. If not, you exit the function with nothing to do. If there is a column selected, you turn off the redraw on the control using the `SetRedraw` method so that none of the movement is seen by the user. Next, you move the selected column to the release column using the `SetColPosition` method. Once you move the column, you resort the grid by calling the `DoSort` function. Finally, you turn the control's redraw back on so that the control is refreshed to show the user the moved column. If you compile and link your application, you should now be able to grab column headers and move the columns about, as in Figure 9.8.

FIGURE 9.8.

The FlexGrid with reordered columns.

Summary

Today you learned how you can use ActiveX controls in your Visual C++ applications to easily extend your application's functionality. You learned the basics of how ActiveX controls work and how they interact with the containing application. You also learned how you can add an ActiveX control to your development project so that you can use it in your application. You saw how Visual C++ creates C++ classes to encapsulate the ActiveX controls that you add and how you can interact with the control through the exposed methods of these generated C++ classes. You also saw how you can capture events that are generated by the ActiveX control so that you can program your application to react to the events.

Q&A

Q How can I determine what methods are available to me when working with an ActiveX control?

A By examining the C++ classes that Visual C++ builds to encapsulate the control, you can get a good idea of what functionality is available to you. If you have documentation for the control, you can compare it to the C++ class to determine which class method calls which control method. You can examine the events listed for the control in the Class Wizard to determine which events are also available.

Q How can I use the ActiveX controls that were installed on my machine with another application in my Visual C++ applications?

A It depends on how the controls are licensed and what application installed the controls. If the controls were installed by another application development tool, chances are that you have a development license for the control, in which case you should be able to use them in your Visual C++ applications. If the controls were installed by an end-user application, such as Word or Quicken, then odds are that you have only a runtime license for the control. If you want to use these controls in your own applications, you need to contact the control developer to acquire a development license for the controls.

Q Because the FlexGrid control does not allow me to enter data directly into the control, how can I let my users enter data into the grid as if they were using a spreadsheet?

A To implement this functionality for the FlexGrid control, you need to add a floating Edit Box control to your window. Your code needs to determine which cell the user wants to edit and float the edit box in front of that cell. This arrangement allows the user to feel as if he is entering data directly into the cell. Another approach is to have a data-entry field outside the grid, much like is used in Excel, into which the user enters the data. You can highlight the cells as the user maneuvers around the Grid control to give the user visceral feedback for her actions.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. How does an ActiveX container call methods in an ActiveX control?
2. How does an ActiveX control trigger events in the container application?
3. What AppWizard option must be selected for ActiveX controls to work properly in a Visual C++ application?
4. How does Visual C++ make it easy to work with ActiveX controls?
5. Why might it be difficult to work with older controls in Visual C++?

Exercise

Modify the application so that the user can double-click a column header and make it the first column in the grid.

WEEK 2

DAY 10

Creating Single Document Interface Applications

Today you will learn a different way of approaching application development with Visual C++ than you have used with the previous days' lessons. Today you will learn how to create Single Document Interface (SDI) applications. An SDI application is a document-centric application that can only work with one document at a time, and can only work with one type of document.

Some good examples of SDI applications are Notepad, WordPad, and Paint. All of these applications can do only one type of task and can only work on one task at a time. WordPad is almost like an SDI version of Word. It's able to perform a large number of the tasks that Word does, but although Word allows you to work on numerous documents at the same time, WordPad limits you to only one document.

Some of the things that you will learn today are

- The Document/View architecture that Visual C++ uses for creating SDI applications.

- How to create an SDI application shell.
- How to separate your data from the visual representation of the data.
- How to encapsulate your data in its own C++ class.
- How to create interaction between the data and the menus.

The Document/View Architecture

When you create an SDI application, more classes are created for an SDI application than for a dialog-style application. Each of these classes serves a specific purpose in how SDI applications operate. Ignoring the About window dialog class, four specific classes make up an SDI application:

- The `CWinApp`-derived class
- The `CFrameView`-derived class
- The `CDocument`-derived class
- The `CView`-derived class

The `CWinApp` class creates all the other components in the application. It is the class that receives all the event messages and then passes the messages to the `CFrameView` and `CView` classes.

The `CFrameView` class is the window frame. It holds the menu, toolbar, scrollbars, and any other visible objects attached to the frame. This class determines how much of the document is visible at any time. Very little (if any) of your programming efforts on SDI applications will require making any modifications or additions to either of these first two classes.

The `CDocument` class houses your document. This class is where you will build the data structures necessary to house and manipulate the data that makes up your document. This class receives input from the `CView` class and passes display information to the `CView` class. This class is also responsible for saving and retrieving the document data from files.

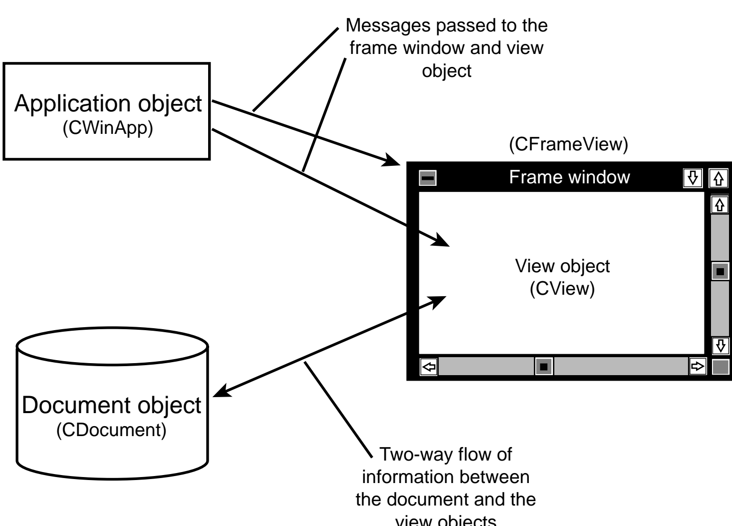
The `CView` class is the class that displays the visual representation of your document for the user. This class passes input information to the `CDocument` class and receives display information from the `CDocument` class. Most of the coding that you will do for this class consists of drawing the document for the user and handling the input from the user. The `CView` class has several descendent classes that can be used as the ancestor for the view class. These descendent classes are listed in Table 10.1.

TABLE 10.1. THE `CView` DESCENDENT CLASSES.

Class	*Description*
`CEditView`	Provides the functionality of a edit box control. Can be used to implement simple text-editor functionality.
`CFormView`	The base class for views containing controls. Can be used to provide form-based documents in applications.
`CHtmlView`	Provides the functionality of a Web browser. This view directly handles the URL navigation, hyperlinking, and so on. Maintains a history list for browsing forward and back.
`CListView`	Provides list-control functionality in the Document/View architecture.
`CRichEditView`	Provides character and paragraph formatting functionality. Can be used to implement a word-processor application.
`CScrollView`	Provides scrolling capabilities to a `CView` class.
`CTreeView`	Provides tree-control functionality in the Document/View architecture.

All four of these classes work together to make up the full functionality of an SDI application, as shown in Figure 10.1. By taking advantage of this architecture, you can build powerful document-centric applications with relative ease.

FIGURE 10.1.
The Document/View architecture.



> Don't let the term document mislead you. This doesn't mean that you can only create applications such as word processors and spreadsheets. In this situation, the term document refers to the data that is processed by your application, whereas view refers to the visual representation of that data. For instance, the Solitaire application could be implemented as a Document/View application, with the document being the cards and their position in the playing area. In this case, the view is the display of the cards, drawing each card where the document specifies it should be.

Creating an SDI Application

To get a good idea of how the Document/View architecture works, and of how you can use it to build applications, you will build a new version of the drawing application you created on Day 3, "Allowing User Interaction—Integrating the Mouse and Keyboard in Your Application." In this version, the user's drawing will be persistent, which means it is not erased each time another window is placed in front of the application. This version will also be able to save and restore drawings.

Building the Application Shell

To create the application shell for today's application, follow these steps:

1. Create a new AppWizard project. Name the project **`Day10`**.
2. On the first step of the AppWizard, select Single Document.
3. Use the default values on the second step of the AppWizard.
4. On the third step of the AppWizard, uncheck the support for ActiveX Controls.
5. On the fourth step of the AppWizard, leave all the default values. Click the Advanced button.
6. In the Advanced Options dialog, enter a three-letter file extension for the files that your application will generate (for example, `dhc` or `dvp`). Click the Close button to close the dialog and then click Next to move to the next step of the AppWizard.
7. Use the default settings on the fifth step of the AppWizard.
8. On the sixth and final AppWizard step, you can choose the base class on which your view class will be based. Leave the base class as `CView` and click Finish. The AppWizard will generate the application shell.

Creating a Line Class

One of the first issues that you will need to tackle is how to represent your data in the document class. For the drawing application, you have a series of lines. Each line consists of a starting point and ending point. You might think that you can use a series of points for the data representation. If you do this, you also have to make special accommodations for where one series of lines between points ends and the next begins. It makes much more sense to represent the drawing as a series of lines. This allows you to store each individual line that is drawn on the window without having to worry where one set of contiguous lines ends and where the next begins.

Unfortunately, the Microsoft Foundation Classes (MFC) does not have a line object class, although it does have a point object class (`CPoint`). I guess you'll just have to create your own line class by following these steps:

1. In the Class View tab of the workspace pane, select the top-level object in the tree (Day10 classes). Right-click the mouse and select New Class from the pop-up menu.
2. In the New Class dialog, select Generic Class for the class type. Enter `CLine` for the class name and click in the first line in the Base Class list box. Enter `CObject` as the base class, leaving the class access as public, as in Figure 10.2.

FIGURE 10.2.
The New Class Wizard.

3. When you click the OK button to add the `CLine` class, you may be told that the Class Wizard cannot find the appropriate header file for inheriting the `CLine` class from the `CObject` class, as in Figure 10.3. Click on the OK button on this message box.

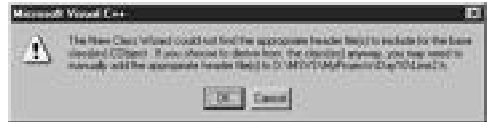


FIGURE 10.3.
Warning about including the base class definition.

> The appropriate header class is already included in the `CLine` class files. Until your compiler complains because it can't find the definition for the `CObject` class, don't worry about this message. However, if you are using a base class that's a bit further down the MFC class hierarchy, you might need to heed this message and add the appropriate header file to the include statements in the class source code file.

Constructing the `CLine` Class

At this time, your `CLine` class needs to hold only two data elements, the two end points of the line that it represents. You want to add those two data elements and add a class constructor that sets both values when creating the class instance. To do this, follow these steps:

1. In the Class View tab of the workspace pane, select the `CLine` class.
2. Right-click the `CLine` class and choose Add Member Variable from the pop-up menu.
3. Enter `CPoint` as the variable type and `m_ptFrom` as the variable name, and mark the access as Private. Click OK to add the variable.
4. Repeat steps 2 and 3, naming this variable `m_ptTo`.
5. Right-click the `CLine` class and choose Add Member Function from the pop-up menu.
6. Leave the function type blank, and enter `CLine(CPoint ptFrom, CPoint ptTo)` for the function declaration. Click OK to add the function.
7. Edit the new function, adding the code in Listing 10.1.

LISTING 10.1. THE `CLine` CONSTRUCTOR.

```
1: CLine::CLine(CPoint ptFrom, CPoint ptTo)
2: {
3:     //Initialize the from and to points
4:     m_ptFrom = ptFrom;
5:     m_ptTo = ptTo;
6: }
```

In this object constructor, you are initializing the from and to points with the points that were passed in to the constructor.

Drawing the CLine Class

To follow correct object-oriented design, your CLine class should be able to draw itself so that when the view class needs to render the line for the user, it can just pass a message to the line object, telling it to draw itself. To add this functionality, follow these steps:

1. Add a new function to the CLine class by selecting Add Member Function from the pop-up menu.
2. Specify the function type as void and the function declaration as Draw(CDC *pDC).
3. Add the code in Listing 10.2 to the Draw function you just added.

LISTING 10.2. THE CLine Draw FUNCTION.

```
1: void CLine::Draw(CDC * pDC)
2: {
3:     // Draw the line
4:     pDC->MoveTo(m_ptFrom);
5:     pDC->LineTo(m_ptTo);
6: }
```

This function is taken almost directly from the application you built a week ago. It's a simple function that moves to the first point on the device context and then draws a line to the second point on the device context.

Implementing the Document Functionality

Now that you have an object to use for representing the drawings made by the user, you can store these CLine objects on the document object in a simple dynamic array. To hold this array, you can add a CObArray member variable to the document class.

The CObArray class is an object array class that dynamically sizes itself to accommodate the number of items placed in it. It can hold any objects that are descended from the CObject class, and it is limited in size only by the amount of memory in the system. Other dynamic array classes in MFC include CStringArray, CByteArray, CWordArray, CDWordArray, and CPtrArray. These classes differ by the type of objects they can hold.

Add the CObArray to CDay10Doc, using the Add Member Variable Wizard and giving it a name of m_oaLines.

Adding Lines

The first functionality that you need to add to the document class is the ability to add new lines. This should be a simple process of getting the from and to points, creating a new line object, and then adding it to the object array. To implement this function, add a new member function to the `CDay10Doc` class, specifying the type as `CLine*` and the declaration as `AddLine(CPoint ptFrom, CPoint ptTo)` with public access. Edit the function, adding the code in Listing 10.3.

LISTING 10.3. THE `CDay10Doc AddLine` FUNCTION.

```
 1: CLine * CDay10Doc::AddLine(CPoint ptFrom, CPoint ptTo)
 2: {
 3:     // Create a new CLine object
 4:     CLine *pLine = new CLine(ptFrom, ptTo);
 5:     try
 6:     {
 7:         // Add the new line to the object array
 8:         m_oaLines.Add(pLine);
 9:         // Mark the document as dirty
10:         SetModifiedFlag();
11:     }
12:     // Did we run into a memory exception?
13:     catch (CMemoryException* perr)
14:     {
15:         // Display a message for the user, giving him or her the
16:         // bad news
17:         AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
18:         // Did we create a line object?
19:         if (pLine)
20:         {
21:             // Delete it
22:             delete pLine;
23:             pLine = NULL;
24:         }
25:         // Delete the exception object
26:         perr->Delete();
27:     }
28:     return pLine;
29: }
```

At first, this function is understandable. You create a new `CLine` instance, passing the from and to points as constructor arguments. Right after that, however, you have something interesting, the following flow control construct:

```
1: try
2: {
3: .
```

```
 4: .
 5: .
 6: }
 7: catch (...)
 8: {
 9: .
10: .
11: .
12: }
```

What is this? This construct is an example of structured exception handling. Some code could fail because of a factor beyond your control, such as running out of memory or disk space, you can place a `try` section around the code that might have a problem. The `try` section should always be followed by one or more `catch` sections. If a problem occurs during the code in the `try` section, the program immediately jumps to the `catch` sections. Each `catch` section specifies what type of exception it handles (in the case of the `AddLine` function, it specifically handles memory exceptions only), and if there is a matching `catch` section for the type of problem that did occur, that section of code is executed to give the application a chance to recover from the problem. If there is no `catch` section for the type of problem that did occur, your program jumps to a default exception handler, which will most likely shut down your application. For more information on structured exception handling, see Appendix A, "C++ Review."

Within the `try` section, you add the new `CLine` instance to the array of line objects. Next, you call the `SetModifiedFlag` function, which marks the document as "dirty" (unsaved) so that if you close the application or open another file without saving the current drawing first, the application prompts you to save the current drawing (with the familiar Yes, No, Cancel message box).

In the `catch` section, you inform the user that the system is out of memory and then clean up by deleting the `CLine` object and the exception object.

Finally, at the end of the function, you return the `CLine` object to the calling routine. This enables the view object to let the line object draw itself.

Getting the Line Count

The next item you will add to the document class is a function to return the number of lines in the document. This functionality is necessary because the view object needs to loop through the array of lines, asking each line object to draw itself. The view object will need to be able to determine the total number of lines in the document and retrieve any specific line from the document.

Returning the number of lines in the document is a simple matter of returning the number of lines in the object array, so you can just return the return value from the `GetSize`

method of the `CObArray` class. To implement this function, add a new member function to the `CDay10Doc` class, specifying the type as `int` and the declaration as `GetLineCount` with public access. Edit the function, adding the code in Listing 10.4.

LISTING 10.4. THE `CDay10Doc GetLineCount` FUNCTION.

```
1: int CDay10Doc::GetLineCount()
2: {
3:     // Return the array count
4:     return m_oaLines.GetSize();
5: }
```

Retrieving a Specific Line

Finally, you need to add a function to return a specific line from the document. This is a simple matter of returning the object at the specified position in the object array. To implement this function, add a new member function to the `CDay10Doc` class, specifying the type as `CLine*` and the declaration as `GetLine(int nIndex)` with public access. Edit the function, adding the code in Listing 10.5.

LISTING 10.5. THE `CDay10Doc GetLine` FUNCTION.

```
1: CLine * CDay10Doc::GetLine(int nIndex)
2: {
3:     // Return a pointer to the line object
4:     // at the specified point in the object array
5:     return (CLine*)m_oaLines[nIndex];
6: }
```

Notice that the object being returned had to be cast as a pointer to a `CLine` object. Because the `CObArray` class is an array of `CObjects`, every element that is returned by the array is a `CObject` instance, not a `CLine` object instance.

Showing the User

Now that you have built the capability into the document class to hold the drawing, you need to add the functionality to the view object to read the user's drawing input and to draw the image. The mouse events to capture the user input are almost identical to those you created a week ago. The second part of the functionality that you need to implement

is drawing the image. You will make an addition to a function that already exists in the view object class.

Before adding these functions, you need to add a member variable to the `CDay10View` class to maintain the previous mouse point, just as you did a week ago. Add a member variable to the `CDay10View` class through the workspace pane, specifying the type as `CPoint`, the name as `m_ptPrevPos`, and the access as private.

Adding the Mouse Events

To add the mouse events to capture the user's drawing efforts, open the Class Wizard and add functions to the `CDay10View` class for the `WM_LBUTTONDOWN`, `WM_LBUTTONUP`, and `WM_MOUSEMOVE` event messages. Edit the functions as in Listing 10.6.

LISTING 10.6. THE `CDay10View` MOUSE FUNCTIONS.

```
 1: void CDay10View::OnLButtonDown(UINT nFlags, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Capture the mouse, so no other application can
10:     // grab it if the mouse leaves the window area
11:     SetCapture();
12:     // Save the point
13:     m_ptPrevPos = point;
14:
15:     ///////////////////////
16:     // MY CODE ENDS HERE
17:     ///////////////////////
18:
19:     CView::OnLButtonDown(nFlags, point);
20: }
21:
22: void CDay10View::OnLButtonUp(UINT nFlags, CPoint point)
23: {
24:     // TODO: Add your message handler code here and/or call default
25:
26:     ///////////////////////
27:     // MY CODE STARTS HERE
28:     ///////////////////////
29:
30:     // Have we captured the mouse?
```

continues

LISTING 10.6. CONTINUED

```
31:     if (GetCapture() == this)
32:         // If so, release it so other applications can
33:         // have it
34:         ReleaseCapture();
35:
36:     ///////////////////////
37:     // MY CODE ENDS HERE
38:     ///////////////////////
39:
40:     CView::OnLButtonUp(nFlags, point);
41: }
42:
43: void CDay10View::OnMouseMove(UINT nFlags, CPoint point)
44: {
45:     // TODO: Add your message handler code here and/or call default
46:
47:     ///////////////////////
48:     // MY CODE STARTS HERE
49:     ///////////////////////
50:
51:     // Check to see if the left mouse button is down
52:     if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
53:     {
54:         // Have we captured the mouse?
55:         if (GetCapture() == this)
56:         {
57:             // Get the Device Context
58:             CClientDC dc(this);
59:
60:             // Add the line to the document
61:             CLine *pLine = GetDocument()->AddLine(m_ptPrevPos, point);
62:
63:             // Draw the current stretch of line
64:             pLine->Draw(&dc);
65:
66:             // Save the current point as the previous point
67:             m_ptPrevPos = point;
68:         }
69:     }
70:
71:     ///////////////////////
72:     // MY CODE ENDS HERE
73:     ///////////////////////
74:
75:     CView::OnMouseMove(nFlags, point);
76: }
```

In the OnLButtonDown function, the first thing you do is call the SetCapture function. This function "captures" the mouse, preventing any other applications from receiving any mouse events, even if the mouse leaves the window space of this application. This enables the user to drag the mouse outside the application window while drawing and then drag the mouse back into the application window, without stopping the drawing. All mouse messages are delivered to this application until the mouse is released in the OnLButtonUp function, using the ReleaseCapture function. In the meantime, by placing the GetCapture function in an if statement and comparing its return value to this, you can determine whether your application has captured the mouse. If you capture the mouse, you want to execute the rest of the code in those functions; otherwise, you don't.

In the OnMouseMove function, after you create your device context, you do several things in a single line of code. The line


```
CLine *pLine = GetDocument()->AddLine(m_ptPrevPos, point);
```

creates a new pointer to a CLine class instance. Next, it calls the GetDocument function, which returns a pointer to the document object. This pointer is used to call the document class's AddLine function, passing the previous and current points as arguments. The return value from the AddLine function is used to initialize the CLine object pointer. The CLine pointer can now be used to call the line object's Draw function.

A pointer is the address of an object. It is used to pass an object more efficiently around a program. Passing a pointer to an object, instead of the object itself, is like telling someone that the remote control is "on the couch between the second and third cushion, beside the loose pocket change" instead of handing the remote to the person. Actually, in programming terms, handing the remote to the person requires making an exact copy of the remote and handing the copy to the other person. It is obviously more efficient to tell the person where to find the remote than to manufacture an exact copy of the remote.

The notation -> denotes that the object's functions or properties are accessed through a pointer, as opposed to directly through the object itself with the period (.) notation.

Drawing the Painting

In the view class, the function OnDraw is called whenever the image presented to the user needs to be redrawn. Maybe another window was in front of the application window, the window was just restored from being minimized, or a new document was just loaded from a file. Why the view needs to be redrawn doesn't matter. All you need to worry

about as the application developer is adding the code to the `OnDraw` function to render the document that your application is designed to create.

Locate the `OnDraw` function in the `CDay10View` class and add the code in Listing 10.7.

LISTING 10.7. THE `CDay10View OnDraw` FUNCTION.

```
 1: void CDay10View::OnDraw(CDC* pDC)
 2: {
 3:     CDay10Doc* pDoc = GetDocument();
 4:     ASSERT_VALID(pDoc);
 5:
 6:     // TODO: add draw code for native data here
 7:
 8:     ///////////////////////
 9:     // MY CODE STARTS HERE
10:     ///////////////////////
11:
12:     // Get the number of lines in the document
13:     int liCount = pDoc->GetLineCount();
14:
15:     // Are there any lines in the document?
16:     if (liCount)
17:     {
18:         int liPos;
19:         CLine *lptLine;
20:
21:         // Loop through the lines in the document
22:         for (liPos = 0; liPos < liCount; liPos++)
23:         {
24:             // Get the from and to point for each line
25:             lptLine = pDoc->GetLine(liPos);
26:             // Draw the line
27:             lptLine->Draw(pDC);
28:         }
29:     }
30:
31:     ///////////////////////
32:     // MY CODE ENDS HERE
33:     ///////////////////////
34: }
```

In this function, the first thing you did was find out how many lines are in the document to be drawn. If there aren't any lines, then there is nothing to do. If there are lines in the document, you loop through the lines using a `for` loop, getting each line object from the document and then calling the line object's `Draw` function.

Before you can compile and run your application, you'll need to include the header file for the `Cline` class in the source code file for the document and view classes. To add this to your application, edit both of these files (`Day10Doc.cpp` and `Day10View.cpp`), adding the `Line.h` file to the includes, as shown in Listing 10.8.

LISTING 10.8. THE `CDay10Doc includes`.

```
1: #include "stdafx.h"
2: #include "Day10.h"
3: #include "MainFrm.h"
4: #include "Line.h"
5: #include "Day10Doc.h"
```


At this point, you should be able to compile and run your application, drawing figures in it as shown in Figure 10.4. If you minimize the window and then restore it, or if you place another application window in front of your application window, your drawing should still be there when your application window is visible again (unlike the application you built a week ago).

FIGURE 10.4.
Drawing with your application.

Saving and Loading the Drawing

Now that you can create drawings that don't disappear the moment you look away, it'd be nice if you could make them even more persistent. If you play with the menus on your application, it appears that the Open, Save, and Save As menu entries on the File menu activate, but they don't seem to do anything. The printing menu entries all work, but the entries for saving and loading a drawing don't. Not even the New menu entry works! Well, you can do something to fix this situation.

Deleting the Current Drawing

If you examine the `CDay10Doc` class, you'll see the `OnNewDocument` function that you can edit to clear out the current drawing. Wrong! This function is intended for initializing any class settings for starting work on a new drawing and not for clearing out an existing drawing. Instead, you need to open the Class Wizard and add a function on the `DeleteContents` event message. This event message is intended for clearing the current contents of the document class. Edit this new function, adding the code in Listing 10.9.

LISTING 10.9. THE `CDay10Doc DeleteContents` FUNCTION.

```
 1: void CDay10Doc::DeleteContents()
 2: {
 3:     // TODO: Add your specialized code here and/or call the base class
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the number of lines in the object array
10:     int liCount = m_oaLines.GetSize();
11:     int liPos;
12:
13:     // Are there any objects in the array?
14:     if (liCount)
15:     {
16:         // Loop through the array, deleting each object
17:         for (liPos = 0; liPos < liCount; liPos++)
18:             delete m_oaLines[liPos];
19:         // Reset the array
20:         m_oaLines.RemoveAll();
21:     }
22:
23:     ///////////////////////
24:     // MY CODE ENDS HERE
25:     ///////////////////////
26:
27:     CDocument::DeleteContents();
28: }
```

This function loops through the object array, deleting each line object in the array. Once all the lines are deleted, the array is reset by calling its `RemoveAll` method. If you compile and run your application, you'll find that you can select File | New, and if you decide not to save your current drawing, your window is wiped clean.

Saving and Restoring the Drawing

Adding the functionality to save and restore your drawings is pretty easy to implement, but it might not be so easy to understand. That's okay; you'll spend an entire day on understanding saving and restoring files, also known as serialization, in three days. In the meantime, find the `Serialize` function in the `CDay10Doc` class. The function should look something like

```
 1: void CDay10Doc::Serialize(CArchive& ar)
 2: {
 3:     if (ar.IsStoring())
 4:     {
 5:         // TODO: add storing code here
 6:     }
 7:     else
 8:     {
 9:         // TODO: add loading code here
10:     }
11: }
```

Remove all the contents of this function, and edit the function so that it looks like Listing 10.10.

LISTING 10.10. THE `CDay10Doc Serialize` FUNCTION.

```
 1: void CDay10Doc::Serialize(CArchive& ar)
 2: {
 3:     ///////////////////////
 4:     // MY CODE STARTS HERE
 5:     ///////////////////////
 6:
 7:     // Pass the serialization on to the object array
 8:     m_oaLines.Serialize(ar);
 9:
10:     ///////////////////////
11:     // MY CODE ENDS HERE
12:     ///////////////////////
13: }
```

This function takes advantage of the functionality of the `CObArray` class. This object array will pass down its array of objects, calling the `Serialize` function on each of the objects. This means that you need to add a `Serialize` function to the `CLine` class. Specify it as a `void` function type with the declaration of `Serialize(CArchive& ar)`. Edit the function, adding the code in Listing 10.11.

LISTING 10.11. THE `CLine Serialize` FUNCTION.

```
1: void CLine::Serialize(CArchive &ar)
2: {
3:     CObject::Serialize(ar);
4:
5:     if (ar.IsStoring())
6:         ar << m_ptFrom << m_ptTo;
7:     else
8:         ar >> m_ptFrom >> m_ptTo;
9: }
```

This function follows basically the same flow that the original `Serialize` function would have followed in the `CDay10Doc` class. It uses the I/O stream functionality of C++ to save and restore its contents.

At this point, if you compile and run your application, you expect the save and open functions to work. Unfortunately, they don't (yet). If you run your application and try to save a drawing, a message box will tell you that the application was unable to save the file, as in Figure 10.5.

FIGURE 10.5.
Unable to save drawings.

The reason that you are unable to save your drawing is that Visual C++ must be told that a class should be serializable. To do this, you add one line to the `CLine` class header file and one line to the `CLine` source code file. Open the `CLine` header file (`Line.h`), and add the `DECLARE_SERIAL` line in Listing 10.12 just after the first line of the class definition.

LISTING 10.12. THE `Line.h` EDIT FOR SERIALIZATION.

```
1: class CLine : public CObject
2: {
3:     DECLARE_SERIAL (CLine)
4: public:
5:     CLine(CPoint ptFrom, CPoint ptTo, UINT nWidth, COLORREF crColor);
```

Next, open the `CLine` source code file, and add the `IMPLEMENT_SERIAL` line in Listing 10.13 just before the class constructor functions.

LISTING 10.13. THE `Line.cpp` EDIT FOR SERIALIZATION.

```
 1: // Line.cpp: implementation of the CLine class.
 2: //
 3: //////////////////////////////////////////////////////////////////////
 4:
 5: #include "stdafx.h"
 6: #include "Day10.h"
 7: #include "Line.h"
 8:
 9: #ifdef _DEBUG
10: #undef THIS_FILE
11: static char THIS_FILE[]=__FILE__;
12: #define new DEBUG_NEW
13: #endif
14:
15: IMPLEMENT_SERIAL (CLine, CObject, 1)
16: //////////////////////////////////////////////////////////////////////
17: // Construction/Destruction
18: //////////////////////////////////////////////////////////////////////
19:
20: CLine::CLine()
21: {
22:
23: }
```


Now if you compile and run your application, you should be able to draw your own self-portrait and save it for posterity, as shown in Figure 10.6.

FIGURE 10.6.

My self-portrait.

Interacting with the Menu

Now that you have a working drawing program, it would be nice if the user could choose the color with which she wants to draw. Adding this functionality requires making changes in the CLine class to associate the color with the line and to CDay10Doc to maintain the currently selected color. Finally, you need to add a pull-down menu to select the desired color.

Adding Color to the CLine Class

The changes to the CLine class are fairly straightforward. The first thing that you need to do is to add another member variable to the CLine class to hold the color of each line. Next, you need to modify the class constructor to add color to the list of attributes to be passed in. Third, you need to modify the Draw function to use the specified color. Finally, you need to modify the Serialize function to save and restore the color information along with the point information. To do all these things, follow these steps:

1. Select the CLine class in the Class View tab of the workspace pane. Right-click the mouse and select Add Member Variable from the pop-up menu.
2. Specify the variable type as COLORREF, the name as m_crColor, and the access as private. Click OK to add the variable.
3. Right-click the CLine constructor in the Class View tree. Select Go to Declaration from the pop-up menu.
4. Add COLORREF crColor as a third argument to the constructor declaration.
5. Right-click the CLine constructor in the Class View tree. Select Go to Definition from the pop-up menu.
6. Modify the constructor to add the third argument and to set the m_crColor member to the new argument, as in Listing 10.14.

LISTING 10.14. THE MODIFIED CLine CONSTRUCTOR.

```
1: CLine::CLine(CPoint ptFrom, CPoint ptTo, COLORREF crColor)
2: {
3:     //Initialize the from and to points
4:     m_ptFrom = ptFrom;
5:     m_ptTo = ptTo;
6:     m_crColor = crColor;
7: }
```

7. Scroll down to the Draw function and modify it as in Listing 10.15.

LISTING 10.15. THE MODIFIED `Draw` FUNCTION.

```
 1: void CLine::Draw(CDC * pDC)
 2: {
 3:     // Create a pen
 4:     CPen lpen (PS_SOLID, 1, m_crColor);
 5:
 6:     // Set the new pen as the drawing object
 7:     CPen* pOldPen = pDC->SelectObject(&lpen);
 8:     // Draw the line
 9:     pDC->MoveTo(m_ptFrom);
10:     pDC->LineTo(m_ptTo);
11:     // Reset the previous pen
12:     pDC->SelectObject(pOldPen);
13: }
```


8. Scroll down to the `Serialize` function and modify it as in Listing 10.16.

LISTING 10.16. THE MODIFIED `Serialize` FUNCTION.

```
1: void CLine::Serialize(CArchive &ar)
2: {
3:     CObject::Serialize(ar);
4:
5:     if (ar.IsStoring())
6:         ar << m_ptFrom << m_ptTo << (DWORD) m_crColor;
7:     else
8:         ar >> m_ptFrom >> m_ptTo >> (DWORD) m_crColor;
9: }
```

The only part of any of these steps that should be a surprise is that you are capturing the return value from the `SelectObject` function when you are specifying the pen to use in drawing the lines. You didn't do this last week. The return value from the `SelectObject` function is the pen that was in use before you changed it. This way, you can use the previous pen to restore it to the device context when you are done drawing.

Adding Color to the Document

The changes that you need to make to the `CDay10Doc` class are just slightly more extensive than those made to the `CLine` class. You need to add a member variable to hold the current color and a color table to convert color IDs into RGB values. You need to initialize the current color variable in the `OnNewDocument` function. Then, you need to modify the `AddLine` function to add the current color to the `CLine` constructor. Finally, you add a function to return the current color. That's all that you need to do for now until you start

adding menu message handlers for setting the current color. To do these things, follow these steps:

1. Select the `CDay10Doc` class in the Class View tab on the workspace pane. Right-click the mouse and choose Add Member Variable from the pop-up menu.
2. Specify the variable type as `UINT`, the name as `m_nColor`, and the access as private. Click OK to add the variable.
3. Repeat step 1.
4. Specify the variable type as "`static const COLORREF`," the name as `m_crColors[8]`, and the access as public.
5. Open the `CDay10Doc` source code (`Day10Doc.cpp`) and add the population of the `m_crColors` color table as in Listing 10.17.

LISTING 10.17. THE COLOR TABLE SPECIFICATION.

```
 1:     //}}AFX_MSG_MAP
 2: END_MESSAGE_MAP()
 3:
 4: const COLORREF CDay10Doc::m_crColors[8] = {
 5:     RGB(   0,   0,   0),    // Black
 6:     RGB(   0,   0, 255),    // Blue
 7:     RGB(   0, 255,   0),    // Green
 8:     RGB(   0, 255, 255),    // Cyan
 9:     RGB( 255,   0,   0),    // Red
10:     RGB( 255,   0, 255),    // Magenta
11:     RGB( 255, 255,   0),    // Yellow
12:     RGB( 255, 255, 255)    // White
13: };
14:
15: //////////////////////////////////////////////////////////////////////
16: // CDay10Doc construction/destruction
17:
18: CDay10Doc::CDay10Doc()
19: .
20: .
21: .
22: }
```

6. Scroll down to the `OnNewDocument` function and edit it as in Listing 10.18.

LISTING 10.18. THE MODIFIED `OnNewDocument` FUNCTION.

```
 1: BOOL CDay10Doc::OnNewDocument()
 2: {
 3:     if (!CDocument::OnNewDocument())
 4:         return FALSE;
 5:
 6:     // TODO: add reinitialization code here
 7:     // (SDI documents will reuse this document)
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Initialize the color to black
14:     m_nColor = ID_COLOR_BLACK - ID_COLOR_BLACK;
15:
16:     ///////////////////////
17:     // MY CODE ENDS HERE
18:     ///////////////////////
19:
20:     return TRUE;
21: }
```


7. Scroll down to the `AddLine` function, and modify it as in Listing 10.19.

LISTING 10.19. THE MODIFIED `AddLine` FUNCTION.

```
 1: CLine * CDay10Doc::AddLine(CPoint ptFrom, CPoint ptTo)
 2: {
 3:     // Create a new CLine object
 4:     CLine *pLine = new CLine(ptFrom, ptTo, m_crColors[m_nColor]);
 5:     try
 6:     {
 7:         // Add the new line to the object array
 8:         m_oaLines.Add(pLine);
 9:         // Mark the document as dirty
10:         SetModifiedFlag();
11:     }
12:     // Did we run into a memory exception?
13:     catch (CMemoryException* perr)
14:     {
15:         // Display a message for the user, giving him or her the
16:         // bad news
17:         AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
18:         // Did we create a line object?
19:         if (pLine)
```

continues

LISTING 10.19. CONTINUED

```
20:         {
21:             // Delete it
22:             delete pLine;
23:             pLine = NULL;
24:         }
25:         // Delete the exception object
26:         perr->Delete();
27:     }
28:     return pLine;
29: }
```

8. Add a new member function to the `CDay10Doc` class. Specify the function type as `UINT`, the declaration as `GetColor`, and the access as public.
9. Edit the `GetColor` function, adding the code in Listing 10.20.

LISTING 10.20. THE `GetColor` FUNCTION.

```
1: UINT CDay10Doc::GetColor()
2: {
3:     // Return the current color
4:     return ID_COLOR_BLACK + m_nColor;
5: }
```

In the `OnNewDocument` and the `GetColor` functions, the color is added and subtracted from `ID_COLOR_BLACK`. This is the lowest numbered color menu ID when you add the menu entries. These calculations maintain the variable as a number between 0 and 7, but when working with the menus, they allow comparison with the actual menu IDs.

Modifying the Menu

Now comes the fun part. You need to add a new pull-down menu to the main menu. You need to add menu entries for all the colors in the color table. You need to add message handlers for all the color menu entries. Finally, you need to add event handlers to check the menu entry that is the current color. To do all of this, follow these steps:

1. Select the Resource View tab in the workspace pane. Expand the tree so that you can see the contents of the Menu folder. Double-click the menu resource.
2. Grab the blank top-level menu (at the right end of the menu bar) and drag it to the left, dropping it in front of the View menu entry.
3. Open the properties for the blank menu entry. Specify the caption as `&Color`. Close the properties dialog.

4. Add submenu entries below the Color top-level menu. Specify the submenus in order, setting their properties as specified in Table 10.2. You should wind up with a menu looking like Figure 10.7.

FIGURE 10.7.
The Color menu as designed.

TABLE 10.2. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	`ID`	`ID_COLOR_BLACK`
	`Caption`	`&Black`
Menu Entry	`ID`	`ID_COLOR_BLUE`
	`Caption`	`B&lue`
Menu Entry	`ID`	`ID_COLOR_GREEN`
	`Caption`	`&Green`
Menu Entry	`ID`	`ID_COLOR_CYAN`
	`Caption`	`&Cyan`
Menu Entry	`ID`	`ID_COLOR_RED`
	`Caption`	`&Red`
Menu Entry	`ID`	`ID_COLOR_MAGENTA`
	`Caption`	`&Magenta`
Menu Entry	`ID`	`ID_COLOR_YELLOW`
	`Caption`	`&Yellow`
Menu Entry	`ID`	`ID_COLOR_WHITE`
	`Caption`	`&White`

5. Open the Class Wizard. Select the `CDay10Doc` in the Class Name combo box.
6. Add functions for both the `COMMAND` and `UPDATE_COMMAND_UI` event messages for all the color menu entries.
7. After the final menu entry function has been added, click Edit Code.
8. Edit the Black menu functions as in Listing 10.21.

LISTING 10.21. THE BLACK MENU FUNCTIONS.

```
 1: void CDay10Doc::OnColorBlack()
 2: {
 3:     // TODO: Add your command handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Set the current color to black
10:     m_nColor = ID_COLOR_BLACK - ID_COLOR_BLACK;
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
16:
17: void CDay10Doc::OnUpdateColorBlack(CCmdUI* pCmdUI)
18: {
19:     // TODO: Add your command update UI handler code here
20:
21:     ///////////////////////
22:     // MY CODE STARTS HERE
23:     ///////////////////////
24:
25:     // Determine if the Black menu entry should be checked
26:     pCmdUI->SetCheck(GetColor() == ID_COLOR_BLACK ? 1 : 0);
27:
28:     ///////////////////////
29:     // MY CODE ENDS HERE
30:     ///////////////////////
31: }
```

9. Edit the Blue menu functions as in Listing 10.22. Edit the remaining menu functions in the same way, substituting their menu IDs for `ID_COLOR_BLUE`.

LISTING 10.22. THE BLUE MENU FUNCTIONS.

```
 1: void CDay10Doc::OnColorBlue()
 2: {
 3:     // TODO: Add your command handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Set the current color to blue
```

```
10:     m_nColor = ID_COLOR_BLUE - ID_COLOR_BLACK;
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
16:
17: void CDay10Doc::OnUpdateColorBlue(CCmdUI* pCmdUI)
18: {
19:     // TODO: Add your command update UI handler code here
20:
21:     ///////////////////////
22:     // MY CODE STARTS HERE
23:     ///////////////////////
24:
25:     // Determine if the Blue menu entry should be checked
26:     pCmdUI->SetCheck(GetColor() == ID_COLOR_BLUE ? 1 : 0);
27:
28:     ///////////////////////
29:     // MY CODE ENDS HERE
30:     ///////////////////////
31: }
```


In the first of the two menu functions, the `COMMAND` function, the current color variable is set to the new color. If you add the menu entries in the correct order, their ID numbers are sequential, starting with `ID_COLOR_BLACK`. Subtracting `ID_COLOR_BLACK` from the menu ID should always result in the correct position in the color table for the selected color. For example, the Black color is position 0 in the color table. `ID_COLOR_BLACK` – `ID_COLOR_BLACK` = 0. Blue is position 1 in the color table. Because `ID_COLOR_BLUE` should be one greater than `ID_COLOR_BLACK`, `ID_COLOR_BLUE` — `ID_COLOR_BLACK` = 1.

The second function, the `UPDATE_COMMAND_UI` function, may need a little explaining. The `UPDATE_COMMAND_UI` event is called for each menu entry just before it is displayed. You can use this event message function to check or uncheck the menu entry, based on whether it is the current color. You can also use this event to enable or disable menu entries or make other modifications as necessary. The code in this function

```
pCmdUI->SetCheck(GetColor() == ID_COLOR_BLUE ? 1 : 0);
```

does several things. First, the `pCmdUI` object that is passed in as the only argument is a pointer to a menu object. The `SetCheck` function can check or uncheck the menu entry, depending on whether the argument passed is `1` or `0` (`1` checks, `0` unchecks). The argument portion for the `SetCheck` function is a flow-control construct that can be somewhat

confusing if you haven't spent a large amount of time programming in C/C++. The first half

```
GetColor() == ID_COLOR_BLUE
```

is a simple boolean conditional statement. It results in a true or false result. The portion following this conditional statement

```
? 1 : 0
```

is basically an `if...else` statement in shorthand. If the conditional statement is `true`, then the value is `1`, and if the statement is `false`, the value is `0`. This is a fancy way of placing an `if..else` flow control within the argument to another function.

If you compile and run your application, you should be able to change the color that you are drawing with. When you pull down the color menu, you should see the current drawing color checked on the menu, as in Figure 10.8.

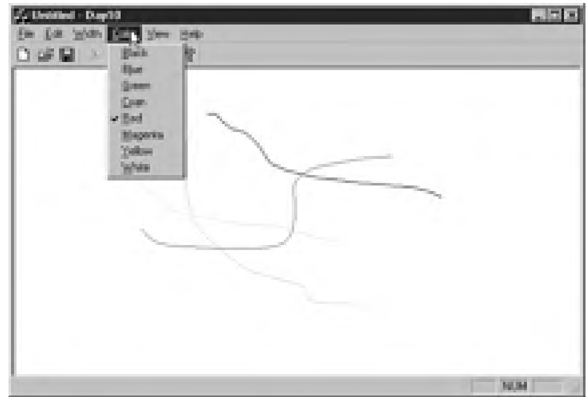


FIGURE 10.8.
Specifying the current color on the menu.

Summary

Whew! What a day! You learned quite a bit today because this was a packed chapter. You initially learned about the SDI style application and about a couple of standard applications that you have probably used that are SDI applications. You next learned about the Document/View architecture that Visual C++ uses for SDI applications. You learned to create a simple class of your own for use in your drawing application. You created a drawing application that can maintain the images drawn using it. You learned how you can save and restore documents in the Document/View architecture. You also learned about the `CObArray` object array class and how you can use it to create a dynamic object array for storing various classes. Finally, you learned how you can check and uncheck menu entries in MFC applications.

Q&A

Q Is there any way that you can reduce the number of `COMMAND` and `UPDATE_COMMAND_UI` functions for the menus?

A Yes, you can send all the color `COMMAND` events to the same function. From there, you can examine the `nID` value (which is passed as an argument) and compare it to the menu IDs to determine which menu is calling the function. As a result, you can write the `COMMAND` function for the color menus as follows:

```
void CDay10Doc::OnColorCommand(UINT nID)
{
    // TODO: Add your command handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Set the current color to blue
    m_nColor = nID - ID_COLOR_BLACK;

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

For the `UPDATE_COMMAND_UI` functions, you can do the same thing, only slightly differently. In this case, you can examine the `pCmdUI->m_nID` value to determine which menu the function is being called for. This makes the `UPDATE_COMMAND_UI` function look like the following:

```
void CDay10Doc::OnUpdateColor(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Determine if the Blue menu entry should be checked
    pCmdUI->SetCheck(GetColor() == pCmdUI->m_nID ? 1 : 0);

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

Q What's the difference between SDI and MDI applications?

A Although SDI applications can perform only one task, MDI (Multiple Document Interface) applications can have multiple documents open at the same time. Plus, in an MDI application, not all document types need be the same. You'll learn more about MDI applications tomorrow.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What does SDI stand for?
2. What functionality is in the view class?
3. What function is called to redraw the document if the window has been hidden behind another window?
4. Where do you place code to clear out the current document before starting a new document?
5. What is the purpose of the document class?

Exercise

Add another pull-down menu to control the width of the pen used for drawing. Give it the following settings:

Menu Entry	*Width Setting*
Very Thin	1
Thin	8
Medium	16
Thick	24
Very Thick	32

Tip: In the pen constructor, the second argument is the width.

WEEK 2

DAY 11

Creating Multiple Document Interface Applications

Today, you will learn how to build Multiple Document Interface (MDI) applications using Visual C++. You will be able to build applications that allow users to work on multiple documents at one time, switching between the windows of the application to do their work. In this chapter, you will learn

- The difference between SDI and MDI applications.
- How to create an MDI application.
- How to send multiple menu entries to a single event-handling function.
- How to add a context menu to a Document/View style application.

What Is an MDI Application?

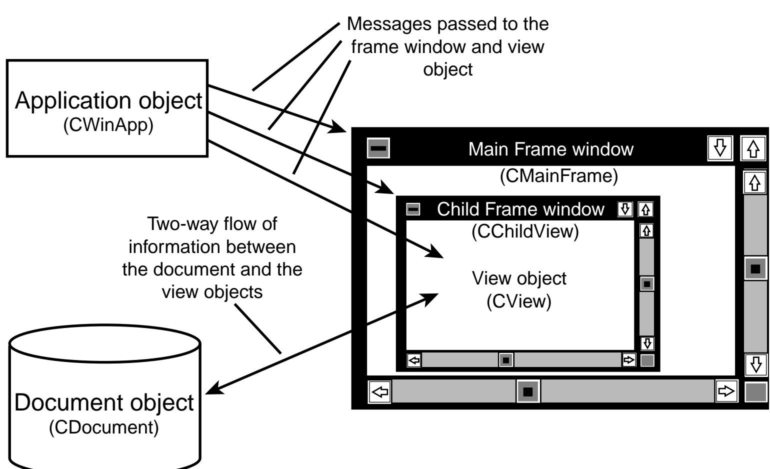
As far as coding an MDI application with Visual C++, there's little difference between creating an SDI and an MDI application. However, when you get

deeper into the two application styles, you'll find quite a few differences. Although an SDI application allows the user to work on only one document at a time, it also normally limits the user to working on a specific type of document. MDI applications not only enable the user to work on multiple documents at the same time, but also MDI applications can allow the user to work on multiple types of documents.

An MDI application uses a window-in-a-window style, where there is a frame window around one or more child windows. This is a common application style with many popular software packages, including Word and Excel.

Architecturally, an MDI application is similar to an SDI application. In fact, with a simple MDI application, the only difference is the addition of a second frame class to the other classes that the AppWizard creates, as shown in Figure 11.1. As you can see, the Document/View architecture is still very much the approach you use for developing MDI applications as well as SDI applications.

FIGURE 11.1.
The MDI Document/View architecture.



When you create an MDI application, you will create just one more class than you created with an SDI application. The classes are

- The `CWinApp` derived class
- The `CMDIFrameWnd` derived class
- The `CMDIChildWnd` derived class
- The `CDocument` derived class
- The `CView` derived class

The two MDI derived classes, `CMDIFrameWnd` (the `CMainFrame` class in your project) and `CMDIChildWnd` (the `CChildFrame` class in your project), are the only two classes that are different from the SDI application that you created.

The first of these two classes, the `CMDIFrameWnd`-derived `CMainFrame`, is the main frame of the application. It provides an enclosed space on the desktop within which all application interaction takes place. This frame window is the frame to which the menu and toolbars are attached.

The second of these two classes, the `CMDIChildWnd`-derived `CChildFrame` class, is the frame that holds the `CView` class. It is the frame that passes messages and events to the view class for processing or display.

In a sense, the functionality of the frame class in the SDI application has been split into these two classes in an MDI application. There is additional support for running multiple child frames with their own document/view class instances at the same time.

Creating an MDI Drawing Program

To get a good understanding of just how alike the Document/View architectures are for the SDI and MDI applications, today you will implement that same drawing application that you created yesterday, only this time as an MDI application.

Building the Application Shell

To create the application shell for today's application, follow these steps:

1. Create a new AppWizard project. Name the project **`Day11`**.
2. On the first step of the AppWizard, select Multiple Documents, as shown in Figure 11.2.

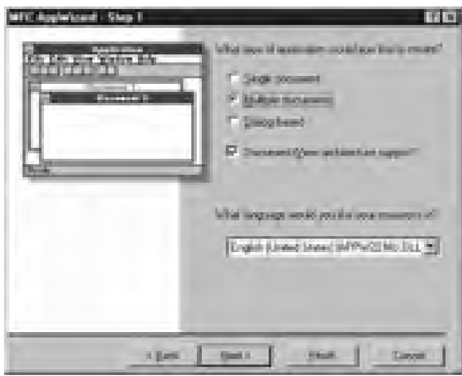


FIGURE 11.2.
Specifying an MDI application.

3. Use the default values on the second step of the AppWizard.
4. On the third step of the AppWizard, uncheck the support for ActiveX Controls.
5. On the fourth step of the AppWizard, leave all the default values. Click the Advanced button.
6. In the Advanced Options dialog, enter a three-letter file extension for the files that your application will generate (for example, `dhc` or `dvp`). Click the Close button to close the dialog and then click Next to move to the next step of the AppWizard.
7. Use the default settings on the fifth step of the AppWizard.
8. On the sixth and final AppWizard step, leave the base class as `CView` and click Finish. The AppWizard generates the application shell.

Building the Drawing Functionality

Because you are creating the same application that you created yesterday, only as an MDI application this time, you need to add the same functionality to the application that you added yesterday. To save time, and to reemphasize how alike these two application architectures are, perform the same steps you did yesterday to create the `CLine` class and add the functionality to the `CDay11Doc` and `CDay11View` classes. Add the support into the `CDay11Doc` and `CLine` classes for selecting colors and widths, but do not add any menu event message handlers or create the color menu. When you finish adding all that functionality, you should have an application in which you can open multiple drawings, all drawing with only the color black.

Because you haven't created the menus yet, and the color initialization uses the color menu IDs, you will probably have to hard-code the initialization of the color to `0` to get your application to compile. Once you add the color menu, the menu IDs should have been added, so you will be able to return to using the IDs in your code. For the time being, change the line of code in the `OnNewDocument` function in the `CDay11Doc` class from

```
m_nColor = ID_COLOR_BLACK - ID_COLOR_BLACK;
```

to

```
m_nColor = 0;
```

You will also need to make the same sort of change to the `GetColor` function because it uses one of the color menu IDs also.

Adding Menu Handling Functionality

Now that you've got all the functionality in your application, you would probably like to add the color menu so you can use all those available colors in your drawings. When you expand the Resource View tree and look in the Menu folder, you'll find not one, but two menus defined. Which one do you add the color menu to?

The `IDR_MAINFRAME` menu is the menu that is available when no child windows are open. If you run your application and close all child windows, you'll see the menu change, removing all the menus that apply to child windows. Once you open another document, either by creating a new document or by opening an existing document, the menu changes back, returning all the menus that apply to the documents.

The `IDR_DAY11TYPE` menu is the menu that appears when a child window is open. This menu contains all the functions that apply to documents. Therefore, this is the menu that you need to add the color menu to. Add the color menu by following the same directions as yesterday, using the same menu properties.

Once you add all the menus, you need to add the menu event handlers. Today, you are going to take a different approach to implementing the menu event handlers than you did yesterday. The Q&A section at the end of yesterday's chapter had a discussion of using a single event-handler function for all the color menus. That is what you are going to implement today. Unfortunately, the Class Wizard doesn't understand how to route multiple menu event messages to the same function correctly, so you're going to implement this yourself by following these steps:

1. Open the `Day11Doc.h` header file.
2. Scroll down toward the bottom of the header file until you find the protected section where the `AFX_MSG` message map is declared (search for `//{{AFX_MSG(CDay11Doc)`).
3. Add the function declarations in Listing 11.1 before the line that you searched for. (The string that you searched for is the beginning marker for the Class Wizard maintained message map. Anything you place between it and the end marker, `//}}AFX_MSG`, is likely to be removed or corrupted by the Class Wizard.)

LISTING 11.1. THE EVENT-HANDLER DECLARATIONS IN `Day11Doc.h`.

```
.
.
.
 1: #ifdef _DEBUG
 2:     virtual void AssertValid() const;
```

continues

LISTING 11.1. CONTINUED

```
 3:     virtual void Dump(CDumpContext& dc) const;
 4: #endif
 5:
 6: protected:
 7:
 8: // Generated message map functions
 9: protected:
10:     afx_msg void OnColorCommand(UINT nID);
11:     afx_msg void OnUpdateColorUI(CCmdUI* pCmdUI);
12:     //{{AFX_MSG(CDay11Doc)
13:        // NOTE - the ClassWizard will add and remove member functions
➥here.
14:        //    DO NOT EDIT what you see in these blocks of generated
➥code !
15:     //}}AFX_MSG
16:     DECLARE_MESSAGE_MAP()
17: private:
18:     UINT m_nColor;
19:     CObArray m_oaLines;
20: };
```

4. Open the `Day11Doc.cpp` source-code file.
5. Search for the line `BEGIN_MESSAGE_MAP` and add the lines in Listing 11.2 just after it. It's important that this code be between the `BEGIN_MESSAGE_MAP` line and the `//{{AFX_MSG_MAP` line. If these commands are between the `//{{AFX_MSG_MAP` and `//}}AFX_MSG_MAP` lines, then the Class Wizard will remove or corrupt them.

LISTING 11.2. THE EVENT-HANDLER MESSAGE MAP ENTRIES IN `Day11Doc.cpp`.

```
 1: ///////////////////////////////////////////////////////////////////////
 2: // CDay11Doc
 3:
 4: IMPLEMENT_DYNCREATE(CDay11Doc, CDocument)
 5:
 6: BEGIN_MESSAGE_MAP(CDay11Doc, CDocument)
 7:     ON_COMMAND_RANGE(ID_COLOR_BLACK, ID_COLOR_WHITE, OnColorCommand)
 8:     ON_UPDATE_COMMAND_UI_RANGE(ID_COLOR_BLACK, ID_COLOR_WHITE,
➥OnUpdateColorUI)
 9:     //{{AFX_MSG_MAP(CDay11Doc)
10:         // NOTE - the ClassWizard will add and remove mapping macros
➥here.
```

```
11:         //    DO NOT EDIT what you see in these blocks of generated
➥code!
12:     //}}AFX_MSG_MAP
13: END_MESSAGE_MAP()
14:
15: const COLORREF CDay11Doc::m_crColors[8] = {
16:     RGB(   0,   0,   0),    // Black
17:     RGB(   0,   0, 255),    // Blue
18: .
19: .
20: .
```

6. Scroll to the bottom of the file and add the two event message handler functions in Listing 11.3.

LISTING 11.3. THE COLOR MENU EVENT-HANDLER FUNCTIONS.

```
 1: void CDay11Doc::OnColorCommand(UINT nID)
 2: {
 3:     // Set the current color
 4:     m_nColor = nID - ID_COLOR_BLACK;
 5: }
 6:
 7: void CDay11Doc::OnUpdateColorUI(CCmdUI* pCmdUI)
 8: {
 9:     // Determine if the menu entry should be checked
10:     pCmdUI->SetCheck(GetColor() == pCmdUI->m_nID ? 1 : 0);
11: }
```


In Listing 11.1, the two function declarations that you added are specified as event message handlers by the `afx_msg` function type declarations. These type of function declarations need to have protected access. Otherwise, they are virtually identical to any other class member function declaration.

In Listing 11.2, the two message map entries, `ON_COMMAND_RANGE` and `ON_UPDATE_COMMAND_UI_RANGE`, are standard message map entries, but the Class Wizard does not support or understand them. If you examine the message map entries from the previous day's applications, you will notice that there are `ON_COMMAND` and `ON_UPDATE_COMMAND_UI` message map entries. These macros have two arguments, the message ID and the event-handler function name that should be called for the event message. These new message map entries function in the same way, but they have two event ID arguments instead of one. The two event ID arguments mark the two ends of a range of

event IDs that should be passed to the function specified. These two event IDs should be the first and last menu entries you created when building the color menu.

> The message map is a mechanism used by Visual C++ and MFC to easily specify event messages and the functions that should be called to handle the event. These message-map commands are converted by the Visual C++ compiler into a fast and efficient map for calling the appropriate event functions when a message is received by the application. Whenever you add a function through the Class Wizard, you are not only adding the function to the code, but you are also adding an entry into the message map for that class.

When you use the `ON_COMMAND_RANGE` message-map entry, the event message ID is automatically passed as an argument to the event-handler function. This allows you to create the function in Listing 11.3 to handle the color selection event messages. If you compile and run your application at this point, you should find that the color selection functionality is all working just as it did yesterday, as shown in Figure 11.3.

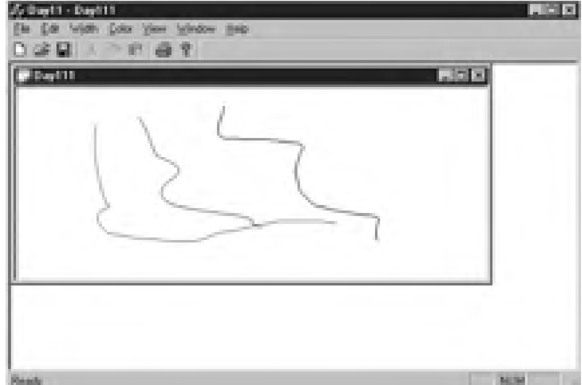


FIGURE 11.3.
Running the MDI application.

Adding a Context Menu

In most Windows applications, you can right-click the mouse and what is known as a context menu, or pop-up menu, appears. Back on Day 6, "Creating Menus for Your Application," you implemented a simple pop-up menu. However, there is a mechanism for creating and using these context menus when Windows thinks that the menu should be opened. This process allows you to add context menus that behave more consistently

with other Windows applications (and if Microsoft changes how the context menus are triggered with a new version of Windows, yours will still behave according to the Windows standard).

An event message `WM_CONTEXTMENU` is passed to the event queue when the right mouse button is released or when the context menu button is pressed (if you have a newer Windows-enabled keyboard with the context menu button). If you place an event-handler function on the `WM_CONTEXTMENU` event message, you can display a pop-up menu with confidence that you are showing it at the appropriate time.

To add the context menu to your application, you create a new menu for use as the context menu. To do this, follow these steps:

1. In the Resource View tab on the workspace pane, right-click the Menu folder.
2. Select Insert Menu from the pop-up menu (or should I say context menu).
3. Select the new menu (still in the workspace pane), open its properties dialog, and name the menu `IDR_CONTEXTMENU`.
4. In the Menu Designer, specify the top-level menu caption as a single space. This causes Visual C++ to add the first entry in the drop-down portion of the menu.
5. In the first drop-down menu entry, specify the caption as `&Width` and check the Pop-up check box. (This causes the ID combo box to be disabled and an arrow to display beside the caption, along with another menu entry to the right of the menu entry you are modifying.)
6. Do not add any menu entries into the Width cascading menu at this time (that is left for an exercise at the end of the chapter). Instead, select the menu entry below the Width entry and open its properties dialog. Specify the caption as `&Colors` and check the Pop-up check box.
7. In the colors cascading menu, add the color menu entries as you did for the `IDR_DAY11TYPE` menu, using the same property settings. You can select the ID from the drop-down list of IDs, if you would rather search for them instead of type. When you finish, your menu should look like the one in Figure 11.4.
8. Select the Class View tab in the workspace pane.
9. Select the CDay11View class. Open the Class Wizard by selecting View | ClassWizard from the menu.

FIGURE 11.4.
The context menu design.

10. Add a function for the `WM_CONTEXTMENU` event message on the `CDay11View` class.
11. Edit the function, adding the code in Listing 11.4.

LISTING 11.4. THE `CDay11View OnContextMenu` FUNCTION.

```
 1: void CDay11View::OnContextMenu(CWnd* pWnd, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     CMenu menu;
10:
11:     // Load the context menu
12:     menu.LoadMenu(IDR_CONTEXTMENU);
13:     // Get the first sub menu (the real menu)
14:     CMenu *pContextMenu = menu.GetSubMenu(0);
15:
16:     // Display the context menu for the user
17:     pContextMenu->TrackPopupMenu(TPM_LEFTALIGN ¦
18:         TPM_LEFTBUTTON ¦ TPM_RIGHTBUTTON,
19:         point.x, point.y, AfxGetMainWnd());
20:
21:     ///////////////////////
```

```
22:     // MY CODE ENDS HERE
23:     ///////////////////////
24: }
```

This code should all look familiar to you from what you learned on Day 6. If you compile and run your application now, you should be able to click your right mouse button on the child window and change your drawing color from the context menu that opened, as shown in Figure 11.5.

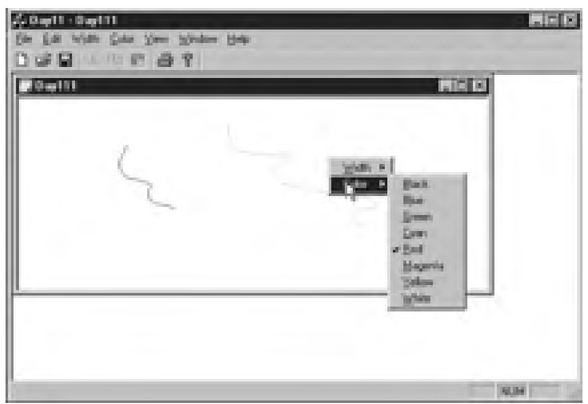


FIGURE 11.5.
Using the context menu to change drawing colors.

Summary

That wasn't too bad; was it? After yesterday, you probably needed the easy day today, along with all the review of what you did yesterday to help it all sink in. But you did get to learn some new things today. You learned about MDI applications, what they are, and how they differ from SDI applications. You learned how you could take a series of menus and use a single event-handler function for all of them. You also learned how you can create a menu specifically for use as a pop-up context menu and how you can integrate it into an MDI application.

Q&A

Q Because it's basically the same code to create an MDI or SDI application, why would I want to create an SDI application? Why wouldn't I want to make all my applications MDI applications?

A It depends on the application and how it's going to be used. You probably use both types of applications on a daily basis. If you are writing a memo or working on a

spreadsheet, you are probably using an MDI application. If you are browsing the World Wide Web, your Web browser is most likely an SDI application. A simple text editor such as Notepad would probably be more difficult for the user as an MDI style application, but as an SDI application, it's just about right (for the task it handles). Certain applications make more sense implemented as an SDI application than as an MDI application. You need to think through how your application is going to be used and determine which model it's more suited for.

Q Some entries on my color menu are changing to the wrong color. How can I determine the problem?

A The problem is that the color menu IDs are probably not in sequential order or are out of order. You can check them by right-clicking on the Day11 resources in the Resource View tab of the workspace pane. Select Resource Symbols from the pop-up menu to display a list of the IDs and the numbers assigned to them in alphabetical order. Start with the Black ID and make sure that the numbers increase by 1 without skipping any numbers. Be sure to check these IDs in the order that the colors appear on the menu (and in the color table in the `Day11Doc.cpp` file), not in the alphabetical order in which they are displayed in this list. If you find some errors, you have to close Visual C++ and open the `Resource.h` file in a text editor to renumber the IDs correctly. Once you make the corrections (be sure to delete any duplicates), save your corrections, restart Visual C++, and recompile your application. The color menu should work correctly.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What are the five base classes that are used in MDI applications?
2. Why do you have to place the `ON_COMMAND_RANGE` message map entry outside the section maintained by the Class Wizard?
3. What argument does `ON_COMMAND_RANGE` pass to the event function?
4. What event message should you use to display a pop-up menu?

Exercise

Add the pull-down and context menus for the width, using the same pen widths as yesterday.

DAY 12

Adding Toolbars and Status Bars

When you created your SDI and MDI applications, they not only came with default menus already attached, but also they came with simple toolbars to go with the menus. These simple toolbars had the standard set of functions (New, Open, Save, Print, Cut, Copy, and Paste) that are on the toolbars of most Windows applications. Most applications don't limit their toolbars to just this standard selection of functions but have customized toolbars that reflect the specific functionality of the application.

In addition to the toolbars, the SDI and MDI applications have a status bar at the bottom of the frame that provides textual descriptions of the toolbar buttons and menu entries. The status bar also has default areas that display whether the Caps, Num, and Scroll Lock keys are on.

Today, you will learn

- How to design your own toolbar.
- How to attach your toolbar to the application frame.

- How to show and hide your toolbar with a menu entry.
- How to place a combo box on your toolbar.
- How to display descriptions of your toolbar entries in the status bar.
- How to add your own status bar elements.

Toolbars, Status Bars, and Menus

One of the driving intentions behind the development of Graphical User Interfaces (GUI) such as Windows was the goal of making computers easier to use and learn. In the effort to accomplish this goal, GUI designers stated that all applications should use a standard set of menus and that the menus should be organized in a standardized manner. When Microsoft designed the Windows operating system, it followed this same philosophy, using a standard set of menus organized in a standard order on most of its applications.

A funny thing happened once Windows became widely used. The application designers found that new users still had a difficult time learning new applications and that advanced users found the menus cumbersome. As a result, the application designers invented toolbars as one solution to both problems.

A toolbar is a small band attached to the window frame or a dialog window that is floating independent of the application frame. This band (or dialog) has a number of small buttons containing graphic images that can be used in place of the menus. The application designers place the most commonly used functions for their applications on these toolbars and do their best to design graphical images that illustrate the functions the buttons serve.

Once advanced users learned what each of the toolbar buttons do, the toolbars were a hit. However, novice users still had problems learning what the toolbar does. As a result, the application designers went back to the drawing board to come up with ways to help the new user learn how use the toolbar buttons.

One of the solutions was to use the information bar that many of them had begun placing at the bottom of application windows to provide detailed descriptions of both menu entries and toolbar buttons. One of the other solutions was to provide a little pop-up window with a short description of the button that appears whenever the mouse is positioned over the button for more than a couple of seconds. The first of these solutions became known as the status bar, and the second became known as tooltips. Both solutions are in common practice with most Windows applications today.

If you want to design and use your own toolbars and status bars in your applications, you might think that Visual C++ provides plenty of support for your efforts and even makes

it easy to implement. After all, Microsoft's own application developers have been in the forefront of developing these elements, and most, if not all, of Microsoft's Windows applications are developed using its own Visual C++. Well, you are correct in making that assumption, and today, you'll learn how to create your own custom toolbars and status bars for your applications.

Designing a Toolbar

For learning how to create your own toolbar, you will modify the application that you created on Day 10, "Creating Single Document Interface Applications," the SDI drawing application, to add a toolbar for selecting the color to use in drawing.

> Although the sample application you are working with today is an extension to the application you built on Day 10, all file and class names have been changed from Day10 to Toolbar. If you are making the changes in the Day 10 project, then when the following text specifies that you make changes to the `CToolbarDoc` class, you should make the changes to the `CDay10Doc` class. Likewise, when you are asked to edit the `Toolbar.rc` file, you can edit the `Day10.rc` file.

If all you want to do is add a few additional toolbar buttons to the default toolbar that the AppWizard creates when you start a new SDI or MDI application, you can pull up the toolbar in the Visual C++ designer through the Resource View in the workspace pane and begin adding new buttons. Just as in the Menu Designer, the end of the toolbar always has a blank entry, waiting for you to turn it into another toolbar button, as shown in Figure 12.1. All you have to do is select this blank button, drag it to the right if you want a separator between it and the button beside it, or drag it to a different position if you want it moved. After you have the button in the desired location, you paint an icon on the button that illustrates the function that the button will trigger. Finally, double-click the button in the toolbar view to open the button's properties dialog and give the button the same ID as the menu that it will trigger. The moment that you compile and run your application, you will have a new toolbar button that performs a menu selection that you chose. If you want to get rid of a toolbar button, just grab it on the toolbar view, and drag it off the toolbar.

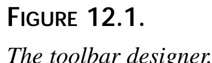


FIGURE 12.1.
The toolbar designer.

Creating a New Toolbar

To insert a new toolbar, right-click on the Toolbar folder and select Insert Toolbar from the pop-up menu. This creates an empty toolbar with a single blank button. As you start drawing an icon on each of the blank buttons in the toolbar, another blank button is added on the end.

For use in your drawing application, fill eight buttons with the eight colors available in the drawing application.

Once you draw icons on each of the buttons in your toolbar, double-click on the first button in the toolbar view. This should open the toolbar button properties dialog. In the ID field, enter (or select from the drop-down list) the ID of the menu that this toolbar button should trigger. In the Prompt field, enter the description that should appear in the status bar for this toolbar button. (If you entered a prompt for the menu, then this field is automatically populated with the menu description.) At the end of the status bar description, add `\n` and a short description to appear in the tooltips for the toolbar button.

In C/C++, the `\n` string is a shorthand notation for "begin a new line." In the prompt for toolbar buttons and menu entries, this string is used to separate the status bar descriptions of the menu entries and the tooltips pop-up prompt that appears when the mouse is held over a toolbar button for a few seconds. The first line of the prompt is used for the status bar description, and the second line is used for the tooltips description. The tooltips description is only used with the toolbars, so there's no reason to add this for menu entries that will have no toolbar equivalents.

For example, for the black button on the toolbar that you are creating for your drawing application, enter an ID of **`ID_COLOR_BLACK`** and a prompt of **`Black drawing color\nBlack`**, as shown in Figure 12.2.

FIGURE 12.2.
The toolbar button properties dialog.

Once you finish designing your toolbar and have icons on all of your buttons with the properties set for each button, you will change the toolbar ID. In the workspace pane, right-click the new toolbar that you just added and open its properties dialog. Change the toolbar ID to a descriptive name.

As an example, for the color toolbar that you created for your drawing application, change the toolbar ID to `IDR_TBCOLOR`.

Attaching the Toolbar to the Application Frame

In the previous SDI and MDI applications, you didn't add any functionality that required you to touch the frame window. Well, because the toolbar is attached to the frame, you'll have to begin adding and modifying code in that module. If you open the `CMainFrame` class to the `OnCreate` function, you'll see where it's creating the existing toolbar and then later in this function where the toolbar is being attached to the frame.

Before you can add your toolbar to the application frame, you need to add a variable to the `CMainFrame` class to hold the new toolbar. This variable of type `CToolBar` should be protected in accessibility.

To add your color toolbar to your draw application, right-click the `CMainFrame` class in the Class View tab of the workspace pane. Select Add Member Variable from the pop-up menu, and specify the variable type as `CToolBar`, the name as `m_wndColorBar`, and the access as protected.

After you add a variable for your toolbar, you need to add some code in the `OnCreate` function in the `CMainFrame` class to add the toolbar and attach it to the frame. Make the modifications in Listing 12.1 to add the color toolbar to your drawing application.

LISTING 12.1. THE MODIFIED `CMainFrame.OnCreate` FUNCTION.

```
 1:  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
 2: {
 3:     if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
 4:         return -1;
 5:
 6:     if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
        ➥WS_CHILD ¦ WS_VISIBLE ¦ CBRS_TOP
 7:         ¦ CBRS_GRIPPER ¦ CBRS_TOOLTIPS ¦ CBRS_FLYBY ¦
        ➥CBRS_SIZE_DYNAMIC) ¦¦
 8:         !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
 9:     {
10:         TRACE0("Failed to create toolbar\n");
11:         return -1;      // fail to create
12:     }
13:
14:     ///////////////////////
15:     // MY CODE STARTS HERE
16:     ///////////////////////
17:
18:     // Add the color toolbar
19:     int iTBCtlID;
20:     int i;
21:
22:     // Create the Color Toolbar
23:     if (!m_wndColorBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD ¦
24:         WS_VISIBLE ¦ CBRS_TOP ¦ CBRS_GRIPPER ¦ CBRS_TOOLTIPS ¦
25:         CBRS_FLYBY ¦ CBRS_SIZE_DYNAMIC) ¦¦
26:         !m_wndColorBar.LoadToolBar(IDR_TBCOLOR))
27:     {
28:         TRACE0("Failed to create toolbar\n");
29:         return -1;      // fail to create
30:     }
31:     // Find the Black button on the toolbar
32:     iTBCtlID = m_wndColorBar.CommandToIndex(ID_COLOR_BLACK);
33:     if (iTBCtlID >= 0)
34:     {
```

```
35:         // Loop through the buttons, setting them to act as radio
               ➥buttons
36:         for (i= iTBCtlID; i < (iTBCtlID + 8); i++)
37:             m_wndColorBar.SetButtonStyle(i, TBBS_CHECKGROUP);
38:     }
39:
40:     ///////////////////////
41:     // MY CODE ENDS HERE
42:     ///////////////////////
43:
44:     if (!m_wndStatusBar.Create(this) ¦¦
45:         !m_wndStatusBar.SetIndicators(indicators,
46:           sizeof(indicators)/sizeof(UINT)))
47:     {
48:         TRACE0("Failed to create status bar\n");
49:         return -1;      // fail to create
50:     }
51:
52:     // TODO: Delete these three lines if you don't want the toolbar to
53:     //  be dockable
54:     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
55:
56:     ///////////////////////
57:     // MY CODE STARTS HERE
58:     ///////////////////////
59:
60:     // Enable docking for the Color Toolbar
61:     m_wndColorBar.EnableDocking(CBRS_ALIGN_ANY);
62:
63:     ///////////////////////
64:     // MY CODE ENDS HERE
65:     ///////////////////////
66:
67:     EnableDocking(CBRS_ALIGN_ANY);
68:     DockControlBar(&m_wndToolBar);
69:
70:     ///////////////////////
71:     // MY CODE STARTS HERE
72:     ///////////////////////
73:
74:     // Dock the Color Toolbar
75:     DockControlBar(&m_wndColorBar);
76:
77:     ///////////////////////
78:     // MY CODE ENDS HERE
79:     ///////////////////////
80:
81:     return 0;
82: }
```

Creating the Toolbar

The first part of the code you added,

```
if (!m_wndColorBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD ¦
    WS_VISIBLE ¦ CBRS_TOP ¦ CBRS_GRIPPER ¦ CBRS_TOOLTIPS ¦
    CBRS_FLYBY ¦ CBRS_SIZE_DYNAMIC) ¦¦
    !m_wndColorBar.LoadToolBar(IDR_TBCOLOR))
```

contains two separate functions that are necessary in creating a toolbar. The first function, `CreateEx`, creates the toolbar itself, whereas the second, `LoadToolBar`, loads the toolbar that you designed in the toolbar designer. The second function, `LoadToolBar`, requires a single argument, the ID for the toolbar that you want to create.

The `CreateEx` function has several arguments that you can pass with the function. The first argument, and the only required argument, is a pointer to the parent window. In this case (which is normally the case), this argument is a pointer to the frame window to which the toolbar will be attached.

The second argument is the style of controls on the toolbar that is to be created. Several toolbar control styles are available for use, some of which have been introduced with the last two versions of Internet Explorer. Table 12.1 lists the available styles.

TABLE 12.1. TOOLBAR CONTROL STYLES.

Style	*Description*
`TBSTYLE_ALTDRAG`	Allows the user to move the toolbar by dragging it while holding down the Alt key.
`TBSTYLE_CUSTOMERASE`	Generates a `NM_CUSTOMDRAW` message when erasing the toolbar and button background, allowing the programmer to choose when and whether to control the background erasing process.
`TBSTYLE_FLAT`	Creates a flat toolbar. Button text appears under the bitmap image.
`TBSTYLE_LIST`	Button text appears to the right of the bitmap image.
`TBSTYLE_REGISTERDROP`	For use in dragging and dropping objects onto toolbar buttons.
`TBSTYLE_TOOLTIPS`	Creates a tooltip control that can be used to display descriptive text for the buttons.
`TBSTYLE_TRANSPARENT`	Creates a transparent toolbar.
`TBSTYLE_WRAPABLE`	Creates a toolbar that can have multiple rows of buttons.

The third argument is the style of the toolbar itself. This argument is normally a combination of window and control bar styles. Normally, only two or three window styles are used, and the rest of the toolbar styles are control bar styles. The list of the normally used toolbar styles appears in Table 12.2.

TABLE 12.2. TOOLBAR STYLES.

Style	*Description*
WS_CHILD	The toolbar is created as a child window.
WS_VISIBLE	The toolbar will be visible when created.
CBRS_ALIGN_TOP	Allows the toolbar to be docked to the top of the view area of the frame window.
CBRS_ALIGN_BOTTOM	Allows the toolbar to be docked to the bottom of the view area of the frame window.
CBRS_ALIGN_LEFT	Allows the toolbar to be docked to the left side of the view area of the frame window.
CBRS_ALIGN_RIGHT	Allows the toolbar to be docked to the right side of the view area of the frame window.
CBRS_ALIGN_ANY	Allows the toolbar to be docked to any side of the view area of the frame window.
CBRS_BORDER_TOP	Places a border on the top edge of the toolbar when the top of the toolbar is not docked.
CBRS_BORDER_BOTTOM	Places a border on the bottom edge of the toolbar when the top of the toolbar is not docked.
CBRS_BORDER_LEFT	Places a border on the left edge of the toolbar when the top of the toolbar is not docked.
CBRS_BORDER_RIGHT	Places a border on the right edge of the toolbar when the top of the toolbar is not docked.
CBRS_FLOAT_MULTI	Allows multiple toolbars to be floated in a single miniframe window.
CBRS_TOOLTIPS	Causes tooltips to be displayed for the toolbar buttons.
CBRS_FLYBY	Causes status bar message text to be updated for the toolbar buttons at the same time as the tooltips.
CBRS_GRIPPER	Causes a gripper to be drawn on the toolbar.

The fourth argument, which you did not provide in your code, is the size of the toolbar borders. This argument is passed as a standard `CRect` rectangle class to provide the length and height desired for the toolbar. The default value is `0` for all of the rectangle dimensions, thus resulting in a toolbar with no borders.

The fifth and final argument, which you also did not provide in your code, is the toolbar's child window ID. This defaults to `AFX_IDW_TOOLBAR`, but you can specify any defined ID that you need or want to use for the toolbar.

Setting the Button Styles

After you create the toolbar, there is a curious bit of code:

```
// Find the Black button on the toolbar
iTBCtlID = m_wndColorBar.CommandToIndex(ID_COLOR_BLACK);
if (iTBCtlID >= 0)
{
    // Loop through the buttons, setting them to act as radio buttons
    for (i= iTBCtlID; i < (iTBCtlID + 8); i++)
        m_wndColorBar.SetButtonStyle(i, TBBS_CHECKGROUP);
}
```

The first line in this code snippet uses the `CommandToIndex` toolbar function to locate the control number of the `ID_COLOR_BLACK` button. If you design your toolbar in the order of colors that you used on the menu, this should be the first control, with a index of `0`. It's best to use the `CommandToIndex` function to locate the index of any toolbar button that you need to alter, just in case it's not where you expect it to be. This function returns the index of the toolbar control specified, and you use this as a starting point to specify the button style of each of the color buttons.

In the loop, where you are looping through each of the eight color buttons on the toolbar, you use the `SetButtonStyle` function to control the behavior of the toolbar buttons. The first argument to this function is the index of the button that you are changing. The second argument is the style of button that you want for the toolbar button specified. In this case, you are specifying that each of the buttons be `TBBS_CHECKGROUP` buttons, which makes them behave like radio buttons, where only one of the buttons in the group can be selected at any time. The list of the available button styles is in Table 12.3.

TABLE 12.3. TOOLBAR BUTTON STYLES.

Style	*Description*
`TBSTYLE_AUTOSIZE`	The button's width will be calculated based on the text on the button.
`TBSTYLE_BUTTON`	Creates a standard push button.
`TBSTYLE_CHECK`	Creates a button that acts like a check box, toggling between the pressed and unpressed state.
`TBSTYLE_CHECKGROUP`	Creates a button that acts like a radio button, remaining in the pressed state until another button in the group is pressed. This is actually the combination of the `TBSTYLE_CHECK` and `TBSTYLE_GROUP` button styles.
`TBSTYLE_DROPDOWN`	Creates a drop-down list button.
`TBSTYLE_GROUP`	Creates a button that remains pressed until another button in the group is pressed.

Style	Description
TBSTYLE_NOPREFIX	The button text will not have an accelerator prefix associated with it.
TBSTYLE_SEP	Creates a separator, making a small gap between the buttons on either side.

Docking the Toolbar

The last thing that you do in the code that you add to the `OnCreate` function in the `CMainFrame` class is the following:

```
// Enable docking for the Color Toolbar
m_wndColorBar.EnableDocking(CBRS_ALIGN_ANY);

EnableDocking(CBRS_ALIGN_ANY);  // (AppWizard generated line)

// Dock the Color Toolbar
DockControlBar(&m_wndColorBar);
```

In the first of these lines, you called the `EnableDocking` toolbar function. This function enables the toolbar for docking with the frame window. The value passed to this toolbar function must match the value passed in the following `EnableDocking` function that is called for the frame window. The available values for these functions are listed in Table 12.4. These functions enable the borders of the toolbar, and the frame window, for docking. If these functions are not called, then you will not be able to dock the toolbar with the frame window. If a specific side is specified in these functions for use in docking, and the sides do not match, you will not be able to dock the toolbar with the frame.

TABLE 12.4. TOOLBAR DOCKING SIDES.

Style	Description
CBRS_ALIGN_TOP	Allows the toolbar to be docked to the top of the view area of the frame window.
CBRS_ALIGN_BOTTOM	Allows the toolbar to be docked to the bottom of the view area of the frame window.
CBRS_ALIGN_LEFT	Allows the toolbar to be docked to the left side of the view area of the frame window.
CBRS_ALIGN_RIGHT	Allows the toolbar to be docked to the right side of the view area of the frame window.
CBRS_ALIGN_ANY	Allows the toolbar to be docked to any side of the view area of the frame window.
CBRS_FLOAT_MULTI	Allows multiple toolbars to be floated in a single miniframe window.
0	The toolbar will not be able to dock with the frame.

The final function that you added was a frame window function, `DockControlBar`, which is passed the address of the toolbar variable. This function physically docks the toolbar to the frame window. Because all of this code appears in the `OnCreate` function for the frame window, the toolbar is docked before the user sees either the window or the toolbar.

Now, after adding all of this code to the `OnCreate` function of the `CMainFrame` class, if you compile and run your application, you'll find a working color toolbar that you can use to select the drawing color, as shown in Figure 12.3.

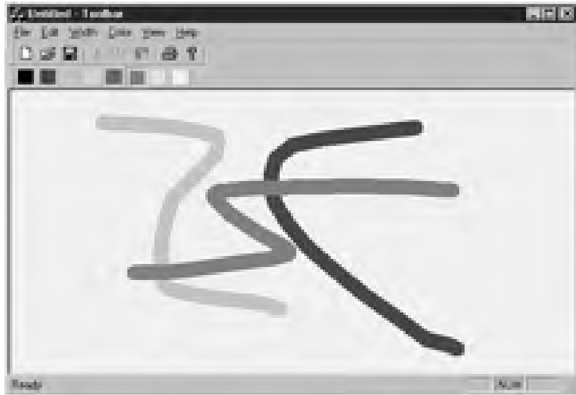


FIGURE 12.3. *The color toolbar on the drawing program.*

Controlling the Toolbar Visibility

Now that you have your color toolbar on the frame of your drawing application, it would be nice to be able to show and hide it just as you can the default toolbar and status bar through the View menu. This is simple enough functionality to add, but it doesn't necessarily work the way you might expect it to.

The first thing you need to do is add a menu entry to toggle the visibility of the color bar. Do this through the Menu Designer, adding a new menu entry on the View menu. Specify the menu properties as shown in Table 12.5.

TABLE 12.5. COLOR BAR MENU PROPERTIES.

Property	*Setting*
ID	`ID_VIEW_COLORBAR`
Caption	`&Color Bar`
Prompt	`Show or hide the colorbar\nToggle ColorBar`

Updating the Menu

To determine whether the toolbar is visible or hidden, you can get the current style of the toolbar and mask out for the `WS_VISIBLE` style flag. If the flag is in the current toolbar style, then the toolbar is visible. By placing this evaluation into the `SetCheck` function in the `UPDATE_COMMAND_UI` event message handler, you can check and uncheck the color bar menu entry as needed.

To add this functionality to your drawing program, add an event handler for the `UPDATE_COMMAND_UI` event message on the `ID_VIEW_COLOR` menu. Be sure to add this event-handler function into the `CMainFrame` class. (You're still making all of your coding changes so far in the frame class.) Edit the event-handler function, adding the code in Listing 12.2.

LISTING 12.2. THE MODIFIED `CMainFrame.OnUpdateViewColorbar` FUNCTION.

```
 1: void CMainFrame::OnUpdateViewColorbar(CCmdUI* pCmdUI)
 2: {
 3:     // TODO: Add your command update UI handler code here
 4:     ///////////////////////
 5:     // MY CODE STARTS HERE
 6:     ///////////////////////
 7:
 8:     // Check the state of the color toolbar
 9:     pCmdUI->SetCheck(((m_wndColorBar.GetStyle() & WS_VISIBLE) != 0));
10:
11:     ///////////////////////
12:     // MY CODE ENDS HERE
13:     ///////////////////////
14: }
```

12

Toggling the Toolbar Visibility

Because the `CToolBar` class is derived from the `CWnd` class (via the `CControlBar` class), you might think that you could call the `ShowWindow` function on the toolbar itself to show and hide the toolbar. Well, you can, but the background for the toolbar will not be hidden along with the toolbar. All the user would notice is the toolbar buttons appearing and disappearing. (Of course, this might be the effect you are after, but your users might not like it.)

Instead, you use a frame window function, `ShowControlBar`, to show and hide the toolbar. This function takes three arguments. The first argument is the address for the toolbar variable. The second argument is a boolean, specifying whether to show the toolbar.

(TRUE shows the toolbar; FALSE hides the toolbar.) Finally, the third argument specifies whether to delay showing the toolbar. (TRUE delays showing the toolbar; FALSE shows the toolbar immediately.)

Once a toolbar is toggled on or off, you need to call another frame window function, RecalcLayout. This function causes the frame to reposition all of the toolbars, status bars, and anything else that is within the frame area. This is the function that causes the color toolbar to move up and down if you toggle the default toolbar on and off.

To add this functionality to your drawing program, add an event handler for the COMMAND event message on the ID_VIEW_COLOR menu. Be sure to add this event-handler function into the CMainFrame class. (You're still making all of your coding changes so far in the frame class.) Edit the event-handler function, adding the code in Listing 12.3.

LISTING 12.3. THE MODIFIED CMainFrame.OnViewColorbar FUNCTION.

```
 1: void CMainFrame::OnViewColorbar()
 2: {
 3:     // TODO: Add your command handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:     BOOL bVisible;
 9:
10:     // Check the state of the color toolbar
11:     bVisible = ((m_wndColorBar.GetStyle() & WS_VISIBLE) != 0);
12:
13:     // Toggle the color bar
14:     ShowControlBar(&m_wndColorBar, !bVisible, FALSE);
15:     // Reshuffle the frame layout
16:     RecalcLayout();
17:
18:     ///////////////////////
19:     // MY CODE ENDS HERE
20:     ///////////////////////
21: }
```

At this point, after compiling and running your application, you should be able to toggle your color toolbar on and off using the View menu.

Adding a Combo Box to a Toolbar

It's commonplace now to use applications that have more than just buttons on toolbars. Look at the Visual C++ Developer Studio, for example. You've got combo boxes that enable you to navigate through your code by selecting the class, ID, and function to edit right on the toolbar. So how do you add a combo box to a toolbar? It's not available in the toolbar designer; all you have there are buttons that you can paint icons on. You can't add a combo box to any toolbar by using any of the Visual C++ wizards. You have to write a little C++ code to do it.

To learn how to add a combo box to a toolbar, you'll add a combo box to the color toolbar you just created. The combo box will be used to select the width of the pen the user will use to draw images. (If you haven't added the support for different drawing widths from the exercise at the end of Day 10, you might want to go back and add that now.)

Editing the Project Resources

To add a combo box to your toolbar, the first thing that you need to do is what Visual C++ was designed to prevent you from having to do. You need to edit the resource file yourself. You cannot do this through the Visual C++ Developer Studio. If you try to open the resource file in the Developer Studio, you will be popped into the Resource View tab of the workspace pane, editing the resource file through the various resource editors and designers. No, you'll have to edit this file in another editor, such as Notepad.

Close Visual C++, the only way to guarantee that you don't write over your changes. Open Notepad and navigate to your project directory. Open the resource file, which is named after the project with a `.rc` filename extension. Once you open this file in Notepad, scroll down until you find the toolbar definitions. (You can search for the word "toolbar.") Once you've found the toolbar definitions, go to the end of the Color toolbar definition and add two separator lines at the bottom of the toolbar definition.

For instance, to make these changes to your drawing application, you need to navigate to the Toolbar project directory and then open the `Toolbar.rc` file. (If you are adding these toolbars to the MDI drawing application, you need to look for the Day11.rc file.) Search for the toolbar section, and then add two `SEPARATOR` lines just before the end of the `IDR_TBCOLOR` section, as shown in Listing 12.4. Once you add these two lines, save the file, exit Notepad, and restart Visual C++, reloading the project.

LISTING 12.4. THE MODIFIED PROJECT RESOURCE FILE (`Toolbar.rc`).

```
 1: ///////////////////////////////////////////////////////////////////////
 2: //
 3: // Toolbar
 4: //
 5:
 6: IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
 7: BEGIN
 8:     BUTTON      ID_FILE_NEW
 9:     BUTTON      ID_FILE_OPEN
10:     BUTTON      ID_FILE_SAVE
11:     SEPARATOR
12:     BUTTON      ID_EDIT_CUT
13:     BUTTON      ID_EDIT_COPY
14:     BUTTON      ID_EDIT_PASTE
15:     SEPARATOR
16:     BUTTON      ID_FILE_PRINT
17:     BUTTON      ID_APP_ABOUT
18: END
19:
20: IDR_TBCOLOR TOOLBAR DISCARDABLE  16, 15
21: BEGIN
22:     BUTTON      ID_COLOR_BLACK
23:     BUTTON      ID_COLOR_BLUE
24:     BUTTON      ID_COLOR_GREEN
25:     BUTTON      ID_COLOR_CYAN
26:     BUTTON      ID_COLOR_RED
27:     BUTTON      ID_COLOR_MAGENTA
28:     BUTTON      ID_COLOR_YELLOW
29:     BUTTON      ID_COLOR_WHITE
30:     SEPARATOR
31:     SEPARATOR
32: END
```

You added these two `SEPARATOR` lines in the toolbar definition so that the second separator can act as a place holder for the combo box that you are going to add to the toolbar. There are two reasons that you had to make this edit by hand and not use the Visual C++ toolbar designer. The first reason is that the toolbar designer would not allow you to add more than one separator to the end of the toolbar. The second reason is that, if you don't add anything else on the end of your toolbar after the separator, the toolbar designer decides that the separator is a mistake and removes it for you. In other words, the Visual C++ toolbar designer does not allow you to add the place holder for the combo box to your toolbar.

Next, you need to add the text strings that you will load into your combo box. To add these strings, you need to open the string table in the Resource View of the workspace pane. Here you find all of the strings that you entered as prompts in various properties dialogs. This table has a number of IDs, the values of those IDs, and textual strings that are associated with those IDs, as shown in Figure 12.4. You'll need to add the strings to be placed into your toolbar combo box in the string table; each line in the drop-down list must have a unique ID and entry in the strings table.

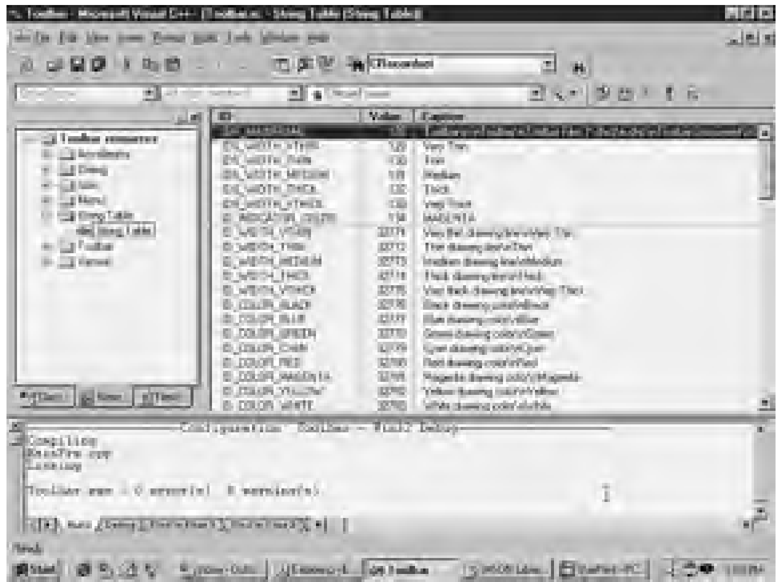


FIGURE 12.4.
The string table editor.

For instance, to add the strings for the combo box that you will be adding to the color toolbar, insert a new string, either by selecting Insert|New String from the menu or by right-clicking the string table and selecting New String from the pop-up menu.

In the String properties dialog, specify a string ID for the string and then enter the string to appear in the drop-down list. Close the properties dialog to add the string. For the strings in the Width combo box that you are going to add to the color toolbar, add the strings in Table 12.6.

TABLE 12.6. WIDTH TOOLBAR COMBO BOX STRINGS.

ID	*Caption*
IDS_WIDTH_VTHIN	Very Thin
IDS_WIDTH_THIN	Thin
IDS_WIDTH_MEDIUM	Medium
IDS_WIDTH_THICK	Thick
IDS_WIDTH_VTHICK	Very Thick

Creating the Toolbar Combo Box

Before you can add the combo box to the color toolbar, you need to create a combo box variable that you can use for the combo box. Because you are not able to add this combo box through any of the designers, you need to add it as a variable to the `CMainFrame` class.

To add the combo box variable to the main frame class for the color toolbar, select the Class View tab in the workspace pane. Right-click the `CMainFrame` class and select Add Member Variable from the pop-up menu. Specify the variable type as `CComboBox`, the name as `m_ctlWidth`, and the access as protected.

Once you add the combo box variable to the main frame class, you need to perform a series of actions, all once the toolbar has been created:

1. Set the width and the ID of the combo box place holder on the toolbar to the width and ID of the combo box.
2. Get the position of the toolbar placeholder and use it to size and position the combo box.
3. Create the combo box, specifying the toolbar as the parent window of the combo box.
4. Load the strings into the drop-down list on the combo box.

To organize this so that it doesn't get too messy, it might be advisable to move the creation of the color toolbar to its own function that can be called from the `OnCreate` function of the main frame class. To create this function, right-click the `CMainFrame` class in the workspace pane and select Add Member Function from the pop-up menu. Specify the function type as `BOOL`, the function description as `CreateColorBar`, and the access as public. Edit the new function, adding the code in Listing 12.5.

LISTING 12.5. THE `CMainFrame CreateColorBar` FUNCTION.

```
 1: BOOL CMainFrame::CreateColorBar()
 2: {
 3:     int iTBCtlID;
 4:     int i;
 5:
 6:     if (!m_wndColorBar.CreateEx(this, TBSTYLE_FLAT,
        ➥WS_CHILD ¦ WS_VISIBLE ¦ CBRS_TOP
 7:         ¦ CBRS_GRIPPER ¦ CBRS_TOOLTIPS ¦ CBRS_FLYBY ¦
            ➥CBRS_SIZE_DYNAMIC) ¦¦
 8:         !m_wndColorBar.LoadToolBar(IDR_TBCOLOR))
 9:     {
10:         TRACE0("Failed to create toolbar\n");
11:         return FALSE;      // fail to create
12:     }
13:     iTBCtlID = m_wndColorBar.CommandToIndex(ID_COLOR_BLACK);
14:     if (iTBCtlID >= 0)
15:     {
16:         for (i= iTBCtlID; i < (iTBCtlID + 8); i++)
17:             m_wndColorBar.SetButtonStyle(i, TBBS_CHECKGROUP);
18:     }
19:     // Add the Combo
20:     int nWidth = 100;
21:     int nHeight = 125;
22:
23:     // Configure the combo place holder
24:     m_wndColorBar.SetButtonInfo(9, IDC_CBWIDTH, TBBS_SEPARATOR,
        ➥nWidth);
25:
26:     // Get the colorbar height
27:     CRect rect;
28:     m_wndColorBar.GetItemRect(9, &rect);
29:     rect.bottom = rect.top + nHeight;
30:
31:     // Create the combo box
32:     m_ctlWidth.Create(WS_CHILD ¦ WS_VISIBLE ¦ WS_VSCROLL ¦
33:         CBS_DROPDOWNLIST, rect, &m_wndColorBar, IDC_CBWIDTH);
34:
35:     //  Fill the combo box
36:     CString szStyle;
37:     if (szStyle.LoadString(IDS_WIDTH_VTHIN))
38:         m_ctlWidth.AddString((LPCTSTR)szStyle);
39:     if (szStyle.LoadString(IDS_WIDTH_THIN))
40:         m_ctlWidth.AddString((LPCTSTR)szStyle);
41:     if (szStyle.LoadString(IDS_WIDTH_MEDIUM))
42:         m_ctlWidth.AddString((LPCTSTR)szStyle);
43:     if (szStyle.LoadString(IDS_WIDTH_THICK))
```

continues

LISTING 12.5. CONTINUED

```
44:         m_ctlWidth.AddString((LPCTSTR)szStyle);
45:     if (szStyle.LoadString(IDS_WIDTH_VTHICK))
46:         m_ctlWidth.AddString((LPCTSTR)szStyle);
47:
48:     return TRUE;
49: }
```

On line 24 in Listing 12.5, you specify that the combo box should be created using the object `ID IDC_CBWIDTH`. This object ID is used to identify the combo box when the combo box sends an event message to the application or when you need to specify what list entry is displayed in the edit field. However, this object ID doesn't exist in your application. Before you can compile the application, you'll need to add this ID to the project resource IDs, just as you did on Day 4, "Working with Timers." To add this ID to your project, select the Resource view in the workspace pane. Select the top of the resource tree and right-click the mouse to trigger the context menu. Select Resource Symbols from the pop-up menu and add the object `ID IDC_CBWIDTH`. Make sure that you add the new object ID with a unique numerical value so that it won't conflict with any other objects in use in your application.

Configuring the Placeholder

After creating the toolbar and configuring all of the toolbar buttons, the first thing you need to do is to configure the separator that is acting as the place holder for the combo box you are about to create. You do this with the `SetButtonInfo` toolbar function, as follows:

```
m_wndColorBar.SetButtonInfo(9, IDC_CBWIDTH, TBBS_SEPARATOR, nWidth);
```

This function takes four arguments. The first argument is the current index of the control in the toolbar—in this case, the tenth control in the toolbar (eight color buttons and two separators). The second argument is the new ID of the toolbar control. This is the ID that will be placed in the event message queue when a control event occurs. The third argument is the type of toolbar control this control should be. The fourth and final argument is somewhat deceptive. If you look at the function documentation, the fourth argument is the new index of the control in the toolbar. This is the position to which the control will be moved. However, if the control is a separator, this argument specifies the width of the control and doesn't move it anywhere. Because this toolbar control is a separator, this argument has the effect of setting it to be as wide as the combo box that you are going to create.

Getting the Toolbar Combo Box Position

Now that you have configured the toolbar separator as the place holder for the combo box, you need to get the position of the combo box place holder on the toolbar so that you can use it to set the position of the combo box:

```
m_wndColorBar.GetItemRect(9, &rect);
rect.bottom = rect.top + nHeight;
```

In the first line, you called the toolbar function `GetItemRect` to get the position and size of the placeholder for the combo box. In the next line, you added the height of the drop-down list to the height that the combo box will eventually be.

Creating the Combo Box

Now that you've got a place holder sized correctly, and you have the position and size for the combo box, it's time to create the combo box. You do this with the `Create` combo box function, as follows:

```
m_ctlWidth.Create(WS_CHILD ¦ WS_VISIBLE ¦ WS_VSCROLL ¦
    CBS_DROPDOWNLIST, rect, &m_wndColorBar, IDC_CBWIDTH);
```

The first argument to the combo box `Create` function is the combo box style. Normally, several style flags are combined to create a combination style value. Table 12.7 lists the flags that you can use in this value.

TABLE 12.7. COMBO BOX STYLES.

Style	*Description*
`WS_CHILD`	Designates this as a child window (required).
`WS_VISIBLE`	Makes the combo box visible.
`WS_DISABLED`	Disables the combo box.
`WS_VSCROLL`	Adds vertical scrolling to the drop-down list.
`WS_HSCROLL`	Adds horizontal scrolling to the drop-down list.
`WS_GROUP`	Groups controls.
`WS_TABSTOP`	Includes the combo box in the tabbing order.
`CBS_AUTOHSCROLL`	Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. This allows the user to enter text wider than the edit control into the combo box.
`CBS_DROPDOWN`	Similar to `CBS_SIMPLE`, but the list is not displayed unless the user selects the icon next to the edit control.

continues

TABLE 12.7. CONTINUED

Style	*Description*
`CBS_DROPDOWNLIST`	Similar to `CBS_DROPDOWN`, but the edit control is replaced with a static-text item displaying the currently selected item in the list.
`CBS_HASSTRINGS`	The owner of the list box is responsible for drawing the list box contents. The list box items consist of strings.
`CBS_OEMCONVERT`	Text entered in the edit control is converted from ANSI to the OEM character set and then back to ANSI.
`CBS_OWNERDRAWFIXED`	The owner of the list box is responsible for drawing the list box contents. The contents of the list are fixed in height.
`CBS_OWNERDRAWVARIABLE`	The owner of the list box is responsible for drawing the list box contents. The contents of the list are variable in height.
`CBS_SIMPLE`	The list box is displayed at all times.
`CBS_SORT`	Automatically sorts the strings in the list box.
`CBS_DISABLENOSCROLL`	List shows a disabled scrollbar when there are not enough items in the list to require scrolling.
`CBS_NOINTEGRALHEIGHT`	Specifies that the combo box is exactly the size specified.

The second argument is the rectangle that the combo box is to occupy. This argument is the position within the parent window—in this case, the toolbar—that the combo box will stay in. It will move with the parent window (the toolbar), staying in this position the entire time.

The third argument is a pointer to the parent window. This is the address of the color toolbar variable.

The fourth argument is the object ID for the combo box.

Populating the Combo Box

The final action that you have to do in creating the combo box on the color toolbar is populate the drop-down list with the available items that the user can select from. You do this with the combination of two functions:

```
if (szStyle.LoadString(IDS_WIDTH_VTHIN))
    m_ctlWidth.AddString((LPCTSTR)szStyle);
```

The first function is a `CString` function, `LoadString`. This function takes a string ID and loads the string matching the ID from the string table. The second function is a combo box function, `AddString`, which adds the string passed in as an argument to the drop-down list. By calling this function combination for each of the elements that should be in the drop-down list, you can populate the combo box from the application string table.

Updating the `OnCreate` Function

After moving all of the code to create the color toolbar to a separate function, you can update the `OnCreate` function so that it calls the `CreateColorBar` function where it used to create the color toolbar, as in Listing 12.6.

LISTING 12.6. THE MODIFIED `CMainFrame.OnCreate` FUNCTION.

```
 1:  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
 2: {
 3:     if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
 4:         return -1;
 5:
 6:     if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
        ➥WS_CHILD ¦ WS_VISIBLE ¦ CBRS_TOP
 7:         ¦ CBRS_GRIPPER ¦ CBRS_TOOLTIPS ¦ CBRS_FLYBY ¦
            ➥CBRS_SIZE_DYNAMIC) ¦¦
 8:         !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
 9:     {
10:         TRACE0("Failed to create toolbar\n");
11:         return -1;      // fail to create
12:     }
13:
14:     ///////////////////////
15:     // MY CODE STARTS HERE
16:     ///////////////////////
17:
18:     // Add the color toolbar
19:     if (!CreateColorBar())
20:     {
21:         TRACE0("Failed to create color toolbar\n");
22:         return -1;      // fail to create
23:     }
24:
25:     ///////////////////////
26:     // MY CODE ENDS HERE
27:     ///////////////////////
28:
29:     if (!m_wndStatusBar.Create(this) ¦¦
30:         !m_wndStatusBar.SetIndicators(indicators,
31:           sizeof(indicators)/sizeof(UINT)))
32:     {
33:         TRACE0("Failed to create status bar\n");
34:         return -1;      // fail to create
35:     }
36:
37:     // TODO: Delete these three lines if you don't want the toolbar to
38:     //  be dockable
39:     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

continues

LISTING 12.6. CONTINUED

```
40:
41:     ///////////////////////
42:     // MY CODE STARTS HERE
43:     ///////////////////////
44:
45:     // Enable docking for the Color Toolbar
46:     m_wndColorBar.EnableDocking(CBRS_ALIGN_ANY);
47:
48:     ///////////////////////
49:     // MY CODE ENDS HERE
50:     ///////////////////////
51:
52:     EnableDocking(CBRS_ALIGN_ANY);
53:     DockControlBar(&m_wndToolBar);
54:
55:     ///////////////////////
56:     // MY CODE STARTS HERE
57:     ///////////////////////
58:
59:     // Dock the Color Toolbar
60:     DockControlBar(&m_wndColorBar);
61:
62:     ///////////////////////
63:     // MY CODE ENDS HERE
64:     ///////////////////////
65:
66:     return 0;
67: }
```

Now when you compile and run your application, you should have a combo box on the end of your color toolbar, as in Figure 12.5. However, the combo box doesn't do anything yet.

FIGURE 12.5.
The color toolbar with a width combo box.

Handling the Toolbar Combo Box Events

Adding an event handler for the combo box is fairly simple, although it does have to be done by hand (because the Class Wizard doesn't even know that the combo box exists). You have to add an `ON_CBN_SELCHANGE` entry into the message map and then add the actual message-handler function into the `CMainFrame` class.

To start with, add the message-handler function by selecting the `CMainFrame` class in the workspace pane and selecting New Member Function from the pop-up menu. Enter the function type as `afx_msg void`, the function definition as `OnSelChangeWidth`, and the access as protected. Edit the new function as in Listing 12.7.

LISTING 12.7. THE `OnSelChangeWidth` FUNCTION.

```
 1: void CMainFrame::OnSelChangeWidth()
 2: {
 3:     // Get the new combo selection
 4:     int nIndex = m_ctlWidth.GetCurSel();
 5:     if (nIndex == CB_ERR)
 6:         return;
 7:
 8:     // Get the active document
 9:     CToolbarDoc* pDoc = (CToolbarDoc*)GetActiveDocument();
10:     // Do we have a valid document?
11:     if (pDoc)
12:         // Set the new drawing width
13:         pDoc->SetWidth(nIndex);
14:
15: }
```


In this function, you first get the current selection from the combo box. Remember that the entries were added in order, and the `CBS_SORT` flag was not specified in the combo box creation, so the selection index numbers should correspond to the widths in the document. As a result, you can get a pointer to the current document instance, using the `GetActiveDocument` function, and then pass the new width to the document using its `SetWidth` function.

For the combo box selection changes to call this message-handler function, you need to add the appropriate entry to the `CMainFrame` message map. Scroll to the top of the `CMainFrame` source code until you find the message map section. Add line 12 in Listing 12.8 to the message map.

LISTING 12.8. THE MODIFIED `CMainFrame` MESSAGE MAP.

```
 1:///////////////////////////////////////////////////////////////////////
 2: // CMainFrame
 3:
 4: IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
 5:
 6: BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
 7:     //{{AFX_MSG_MAP(CMainFrame)
 8:     ON_WM_CREATE()
 9:     ON_COMMAND(ID_VIEW_COLORBAR, OnViewColorbar)
10:     ON_UPDATE_COMMAND_UI(ID_VIEW_COLORBAR, OnUpdateViewColorbar)
11:     //}}AFX_MSG_MAP
12:     ON_CBN_SELCHANGE(IDC_CBWIDTH, OnSelChangeWidth)
13: END_MESSAGE_MAP()
```

This message map entry

```
ON_CBN_SELCHANGE(IDC_CBWIDTH, OnSelChangeWidth)
```

specifies that on combo box selection change events with the object ID of the color toolbar combo box, the `OnSelChangeWidth` function should be called. Now if you compile and run your application, you should be able to change the drawing width with the combo box on the color toolbar.

Updating the Toolbar Combo Box

The one remaining problem with the combo box is that it needs to be updated if the user selects a new value from the menu instead of the combo box. One of the most efficient methods of doing this is to set the current selection in the combo box when any of the menu selections are triggered. This requires a function in the main frame class that can be called from the document class to accomplish this action. All the function in the main frame needs to do is to set the current selection in the combo box.

To implement this function in the main frame, add a new member function to the `CMainFrame` class, specifying the function type as void, the definition as `UpdateWidthCB(int nIndex)`, and the access as public. Once you add this function, edit the function as in Listing 12.9.

LISTING 12.9. THE `CMainFrame.UpdateWidthCB` FUNCTION.

```
1: void CMainFrame::UpdateWidthCB(int nIndex)
2: {
3:     // Set the new selection in the combo box
4:     m_wndColorBar.m_ctlWidth.SetCurSel(nIndex);
5: }
```

This function uses a single combo box function, `SetCurSel`, which sets the current selection in the combo box drop-down list to the entry specified with the index number. The edit control of the combo box is updated with the new selected list entry. If an index number that doesn't exist in the drop-down list is supplied to the combo box, then the function returns an error.

On the document side, you need to call this function in the main frame whenever the appropriate menu event-handling functions are called. Because this could occur in several functions, it makes the most sense to enclose the necessary functionality in a single function. This function needs to get a pointer to the view associated with the document and then, through the view, get a pointer to the frame, which can then be used to call the `UpdateWidthCB` function that you just added to the main frame class.

To add this function to your application, select the `CToolbarDoc` class in the workspace pane, and select Add Member Function from the pop-up menu. Specify void as the function type, `UpdateColorbar(int nIndex)` as the function definition, and private as the function access. Edit the function as in Listing 12.10.

LISTING 12.10. THE `CToolbarDoc.UpdateColorbar` FUNCTION.

```
 1: void CToolbarDoc::UpdateColorbar(int nIndex)
 2: {
 3:     // Get the position of the first view
 4:     POSITION pos = GetFirstViewPosition();
 5:     // Did we get a valid position?
 6:     if (pos != NULL)
 7:     {
 8:         // Get a pointer to the view in that position
 9:         CView* pView = GetNextView(pos);
10:         // Do we have a valid pointer to the view?
11:         if (pView)
12:         {
13:             // Get a pointer to the frame through the view
```

continues

LISTING 12.10. CONTINUED

```
14:             CMainFrame* pFrame = (CMainFrame*)pView-
➥GetTopLevelFrame();
15:             // Did we get a pointer to the frame?
16:             if (pFrame)
17:                 // Update the combo box on the color toolbar
18:                 // through the frame
19:                 pFrame->UpdateWidthCB(nIndex);
20:         }
21:     }
22: }
```

This function traces through the path that you have to follow to get to the application frame from the document class. The first thing that you did was get the position of the first view associated with the document, using the `GetFirstViewPosition` function. A document may have multiple views open at the same time, and this function returns the position of the first of those views.

The next function, `GetNextView`, returns a pointer to the view specified by the position. This function also updates the position variable to point to the next view in the list of views associated with the current document.

Once you have a pointer to the view, you can call the window function, `GetTopLevelFrame`, which returns a pointer to the application frame window. You have to call this function through the view because the document is not descended from the `CWnd` class, although the view is.

Once you have a pointer to the frame window, you can use this pointer to call the function you created earlier to update the combo box on the toolbar. Now if you call this new function from the Width menu command event handlers, as in Listing 12.11, the combo box that you placed on the color toolbar is automatically updated to reflect the currently selected drawing width, regardless of whether the width was selected from the combo box or the pull-down menu.

LISTING 12.11. AN UPDATED WIDTH MENU COMMAND EVENT HANDLER.

```
1: void CToolbarDoc::OnWidthVthin()
2: {
3:     // TODO: Add your command handler code here
4:     // Set the new width
5:     m_nWidth = 0;
6:     // Update the combo box on the color toolbar
7:     UpdateColorbar(0);
8: }
```

Adding a New Status Bar Element

Earlier today, you learned how to specify status bar messages and tooltips for both toolbar buttons and menus. What if you want to use the status bar to provide the user with more substantial information? What if, as in the Visual C++ Developer Studio, you want to provide information about what the user is doing, where he is in the document he is editing, or the mode that the application is in? This information goes beyond the Caps, Num, and Scroll lock keys that Visual C++ automatically reports on the status bar.

It's actually easy to add additional panes to the status bar, as well as take away the panes that are already there. To learn just how easy a change this is, you will add a new pane to the status bar in your drawing application that will display the color currently in use.

Adding a New Status Bar Pane

Before you add a new status bar pane, you need to add a new entry to the application string table for use in the status bar pane. This string table entry will perform two functions for the status bar pane. The first thing it will do is provide the object ID for the status bar pane. You will use this ID for updating the pane as you need to update the text in the pane. The second function this string table entry will perform is size the pane. To size the pane correctly, you need to provide a caption for the string table entry that is at least as wide as the widest string that you will place in the status bar pane.

Add a new string to your application string table, using the same steps you used earlier when adding the text for the combo box you placed on the color toolbar. Specify the string ID as `ID_INDICATOR_COLOR` and the caption as `MAGENTA` (the widest string that you will put into the status bar pane).

A small section in the first part of the main frame source code defines the status bar layout. This small table contains the object IDs of the status bar panes as table elements, in the order in which they are to appear from left to right on the status bar.

To add the color pane to the status bar, add the ID of the color pane to the status bar indicator table definition, just after the message map in the source-code file for the main frame. Place the color pane ID in the table definition in the position that you want it to be on the status bar, as in line 18 of Listing 12.12.

LISTING 12.12. A MODIFIED STATUS BAR INDICATOR TABLE DEFINITION.

```
 1:///////////////////////////////////////////////////////////////////////
 2: // CMainFrame
 3:
 4: IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
 5:
 6: BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
 7:     //{{AFX_MSG_MAP(CMainFrame)
 8:     ON_WM_CREATE()
 9:     ON_COMMAND(ID_VIEW_COLORBAR, OnViewColorbar)
10:     ON_UPDATE_COMMAND_UI(ID_VIEW_COLORBAR, OnUpdateViewColorbar)
11:     //}}AFX_MSG_MAP
12:     ON_CBN_SELCHANGE(IDC_CBWIDTH, OnSelChangeWidth)
13: END_MESSAGE_MAP()
14:
15: static UINT indicators[] =
16: {
17:     ID_SEPARATOR,           // status line indicator
18:     ID_INDICATOR_COLOR,
19:     ID_INDICATOR_CAPS,
20:     ID_INDICATOR_NUM,
21:     ID_INDICATOR_SCRL,
22: };
23:
24:///////////////////////////////////////////////////////////////////////
25: // CMainFrame construction/destruction
```

If you want to drop any of the lock key indicators from the status bar, just remove them from the indicators table definition. If you examine the `OnCreate` function, where the status bar is created (just after the toolbars are created), you'll see where this table is used to create the status bar with the following code:

```
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
```

Once the status bar is created, the `SetIndicators` function is called on the status bar to add the panes as they are defined in the indicators table. The strings associated with the IDs in the indicators table are used to initialize the panes and set their size. If you compile and run your application at this point, you see the new color pane on the status bar with the caption from the string table displayed within.

Setting a Status Bar Pane Text

Once you've added the pane to the status bar, you can let the UPDATE_COMMAND_UI event do all the updating of the pane. All you need to do is add an event handler for this event on the object ID of the pane and use this event to set the pane text. Because the status bar is always visible, the UPDATE_COMMAND_UI event for the panes on the status bar is triggered every time that the application is idle. This means that it is triggered after the application is finished processing just about every keystroke and mouse movement. In almost a week, on Day 18, "Doing Multiple Tasks at One Time—Multitasking," you will learn more about how often and when any tasks that are performed when the application is idle are triggered.

In the event handler, you need to create a string containing the name of the current color (or whatever other text you want to display in the status bar pane). Next, you have to make sure that the pane is enabled. Finally, you need to set the text of the pane to the string that you have created.

To implement this in your application, you need to create an UPDATE_COMMAND_UI event handler. Once again, the Class Wizard does not know about the status bar pane, so you have to create the message handler and add it to the message map yourself. To create the message handler, add a new member function to the document class (CToolbarDoc) with a type of afx_msg void, a definition of OnUpdateIndicatorColor (CCmdUI *pCmdUI), and an access of protected. Edit the newly created function, adding the code in Listing 12.13.

LISTING 12.13. THE OnUpdateIndicatorColor FUNCTION.

```
 1: void CToolbarDoc::OnUpdateIndicatorColor(CCmdUI *pCmdUI)
 2: {
 3:     CString strColor;
 4:
 5:     // What is the current color?
 6:     switch (m_nColor)
 7:     {
 8:     case 0:    // Black
 9:         strColor = "BLACK";
10:         break;
11:     case 1:    // Blue
12:         strColor = "BLUE";
13:         break;
14:     case 2:    // Green
```

continues

LISTING 12.13. CONTINUED

```
15:         strColor = "GREEN";
16:         break;
17:     case 3:    // Cyan
18:         strColor = "CYAN";
19:         break;
20:     case 4:    // Red
21:         strColor = "RED";
22:         break;
23:     case 5:    // Magenta
24:         strColor = "MAGENTA";
25:         break;
26:     case 6:    // Yellow
27:         strColor = "YELLOW";
28:         break;
29:     case 7:    // White
30:         strColor = "WHITE";
31:         break;
32:     }
33:     // Enable the status bar pane
34:     pCmdUI->Enable(TRUE);
35:     // Set the text of the status bar pane
36:     // to the current color
37:     pCmdUI->SetText(strColor);
38: }
```

In this function, you followed three steps exactly: You created a string with the current color name, made sure that the pane was enabled, and set the pane text to the string that you had created.

Now, to make sure that your new message handler is called when it is supposed to be, you need to add an `ON_UPDATE_COMMAND_UI` entry to the message map at the top of the document source code file, as specified in Listing 12.14.

LISTING 12.14. THE MODIFIED `CToolbarDoc` MESSAGE MAP.

```
 1://///////////////////////////////////////////////////////////////////////
 2: // CToolbarDoc
 3:
 4: IMPLEMENT_DYNCREATE(CToolbarDoc, CDocument)
 5:
 6: BEGIN_MESSAGE_MAP(CToolbarDoc, CDocument)
 7:     ON_UPDATE_COMMAND_UI(ID_INDICATOR_COLOR, OnUpdateIndicatorColor)
 8:     //{{AFX_MSG_MAP(CToolbarDoc)
 9:     ON_UPDATE_COMMAND_UI(ID_WIDTH_VTHIN, OnUpdateWidthVthin)
10: .
```

```
11: .
12:     ON_COMMAND(ID_WIDTH_VTHIN, OnWidthVthin)
13:     //}}AFX_MSG_MAP
14: END_MESSAGE_MAP()
```

After adding the message handler and message map entry, you should now be able to compile and run your application and see the color status bar pane automatically updated to reflect the current drawing color, as shown in Figure 12.6.

FIGURE 12.6.
The drawing application with the current color displayed in the status bar.

Summary

12

You learned quite a bit today. (Is this becoming a trend?) You learned how to design and create your own toolbars. Along with learning how to design toolbars, you learned how to specify status bar prompts for the toolbar buttons and menus, along with tooltips text that will display after holding the mouse over toolbar buttons for a couple of seconds. You learned how to create these toolbars and how to attach them to the application frame. You also learned how you can control whether the toolbar is visible from a menu entry.

Next you learned how to place a combo box on a toolbar so that you can provide your application users with the same level of convenience that you have when using many popular software packages. In learning how to add this combo box to the toolbar, you learned how to create a combo box in code, without having to depend on the dialog designers to create combo boxes, and how to populate the combo box drop-down list with text entries. Then, you learned how to tie the combo box into your application by

adding event handlers for the combo box events and how to update the combo box to reflect changes made through the application menus.

Finally, you learned how to add your own panes to the status bar and how you can update the pane to reflect the current status of the application.

Q&A

Q In some applications, toolbars have the option of showing text, as in Internet Explorer. How can I add text to my toolbar buttons?

A Unfortunately, the toolbar designer provides no way to add text to the toolbar buttons. This means that you have to add the text to the buttons in your application code, much in the same way that you had to specify for all of the color toolbar buttons to behave as radio buttons. You use the `SetButtonText` function to set the text on each toolbar button individually. This function takes two arguments: The first is the index number of the button, and the second is the text for the button. If you really want to place text on the toolbar buttons, you also have to resize the toolbar to allow the room for the text to be displayed.

Q I made some changes to the color toolbar in the toolbar designer, and now I get an assertion error every time I try to run my application. What happened?

A The problem is that the toolbar designer found the separators you added to the resource file as place holders for the combo box. The toolbar designer assumed that these were mistakes and removed them for you. The error that you are getting occurs because you are trying to work with a control in the color toolbar that doesn't exist. To fix this problem, reopen the resource file in Notepad and again add the two separators at the end of the color toolbar definition. Then, reload the project into Visual C++ and recompile the application.

Q The combo box on my toolbars looks too big. How can I get it to fit within the toolbar a little better?

A To make the combo box fit within the toolbar like the combo boxes in the Visual C++ Developer Studio, you need to do a couple of things. First, lower the top of the combo box by 3; this places a small border between the top of the combo box and the edge of the toolbar. Next, set the font in the combo box to a smaller font that will fit within the toolbar better. You can experiment with fonts and pitches until you have a font that you like for the combo box in the toolbar.

Q How can I set the text in the first section of the status bar other than by using menu and toolbar prompts?

A You can use `SetWindowText` to set the text in the first pane of the status bar. As a default setting, the first pane in the status bar is a separator that automatically expands to fill the width of the status bar with the other panes right-justified on the bar. The `SetWindowText` function, called on the status bar variable, sets the text in the first pane only. If you want to set the text in any other pane, at any other time than in the `ON_UPDATE_COMMAND_UI` event handler, you can use the `SetPaneText` function. There are two ways that you can set the text in the main part of the status bar. The first is like this:

```
CString myString = "This is my string"
m_wndStatusBar.SetWindowText(myString);
```

The other method is

```
CString myString = "This is my string"
m_wndStatusBar.SetPaneText(0, myString);
```

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

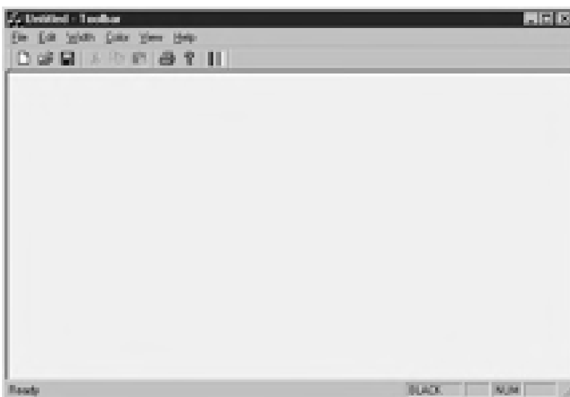
1. How do you tie a toolbar button to a menu entry that triggers that same function?
2. How do you make sure that a toolbar can be docked with the frame window?
3. How can you remove the Num Lock status indicator from the status bar?
4. Why do you have to edit the resource file to add a combo box to a toolbar?

Exercises

1. Add another pane to the status bar to display the current width selected.
2. Add a button to the main toolbar that can be used to toggle the color toolbar on and off, as in Figure 12.7.

FIGURE 12.7.

The color toolbar on/off button.



WEEK 2

DAY 13

Saving and Restoring Work—File Access

Most applications provide the user the option of saving what has been created. The creation can be a word-processing document, a spreadsheet, a drawing, or a set of data records. Today, you will explore how Visual C++ provides you with the means to implement this functionality easily. Today, you will learn

- How Visual C++ uses C++ streams to save information about your application
- How to store your application data in binary files
- How to make your application objects serializable
- How you can store variables of differing data types into a single file

Serialization

There are two parts of serialization. When application data is stored on the system drive in the form of a file, it is called serialization. When the application

state is restored from the file, it is called deserialization. The combination of these two parts makes up the serialization of application objects in Visual C++.

The `CArchive` and `CFile` Classes

Serialization in Visual C++ applications is accomplished through the `CArchive` class. The `CArchive` class is designed to act as an input/output (I/O) stream for a `CFile` object, as shown in Figure 13.1. It uses C++ streams to enable efficient data flow to and from the file that is the storage of the application data. The `CArchive` class cannot exist without a `CFile` class object to which it is attached.

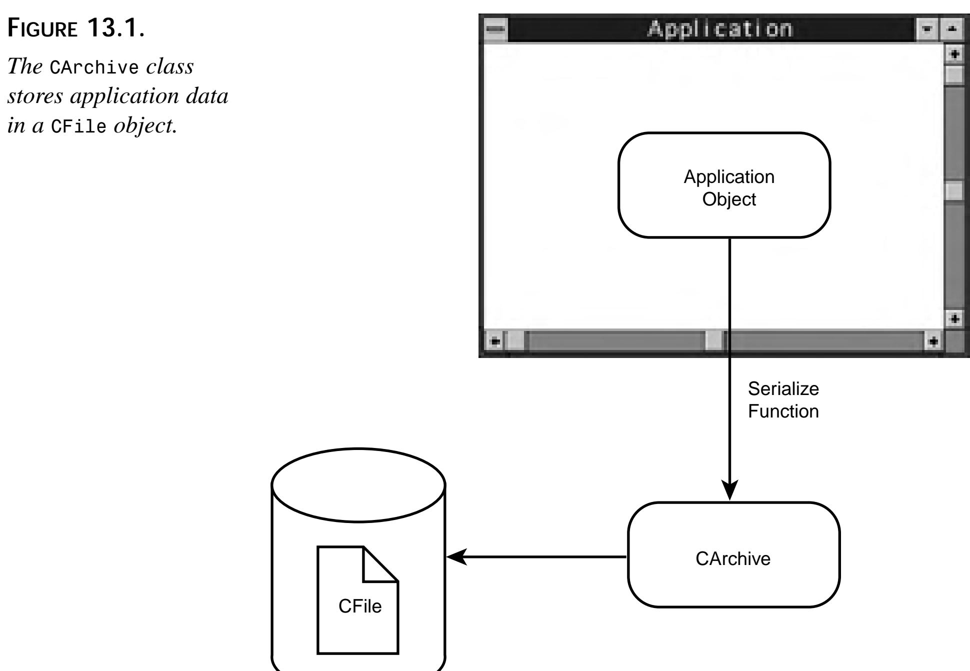


FIGURE 13.1.
The `CArchive` class stores application data in a `CFile` object.

The `CArchive` class can store data in a number of types of files, all of which are descendants of the `CFile` class. By default, the AppWizard includes all the functionality to create and open regular `CFile` objects for use with `CArchive`. If you want or need to work with one of these other file types, you might need to add additional code to your application to enable the use of these different file types.

The Serialize Function

The CArchive class is used in the Serialize function on the document and data objects in Visual C++ applications. When an application is reading or writing a file, the document object's Serialize function is called, passing the CArchive object that is used to write to or read from the file. In the Serialize function, the typical logic to follow is to determine whether the archive is being written to or read from by calling the CArchive IsStoring or IsLoading functions. The return value from either of these two functions determines if your application needs to be writing to or reading from the CArchive class's I/O stream. A typical Serialize function in the view class looks like Listing 13.1.

LISTING 13.1. A TYPICAL Serialize FUNCTION.

```
 1: void CAppDoc::Serialize(CArchive& ar)
 2: {
 3:    // Is the archive being written to?
 4:    if (ar.IsStoring())
 5:    {
 6:       // Yes, write my variable
 7:       ar << m_MyVar;
 8:    }
 9:    else
10:    {
11:       // No, read my variable
12:       ar >> m_MyVar;
13:    }
14: }
```

You can place a Serialize function in any classes you create so that you can call their Serialize function from the document Serialize function. If you place your custom objects into an object array, such as the CObArray that you used in your drawing application for the past three days, you can call the array's Serialize function from the document's Serialize function. The object array will, in turn, call the Serialize function of any objects that have been stored in the array.

Making Objects Serializable

When you created the CLine class on Day 10, "Creating Single Document Interface Applications," you had to add two macros before you could save and restore your drawings. These two macros, DECLARE_SERIAL and IMPLEMENT_SERIAL, include functionality in your classes that are necessary for the Serialize function to work correctly.

Including the DECLARE_SERIAL Macro

You must include the DECLARE_SERIAL macro in your class declaration, as shown in Listing 13.2. The DECLARE_SERIAL macro takes a single argument, the class name. This macro automatically adds to your class some standard function and operator declarations that are necessary for serialization to work correctly.

LISTING 13.2. INCLUDING THE DECLARE_SERIAL MACRO IN THE CLASS DECLARATION.

```
1: class CMyClass : public CObject
2: {
3:     DECLARE_SERIAL (CMyClass)
4: public:
5:     virtual void Serialize(CArchive &ar);
6:     CMyClass();
7:     virtual ~CMyClass();
8: };
```

Including the IMPLEMENT_SERIAL Macro

You need to add the IMPLEMENT_SERIAL macro to the implementation of your class. This macro needs to appear outside any other class functions because it adds the code for the class functions that were declared with the DECLARE_SERIAL macro.

The IMPLEMENT_SERIAL macro takes three arguments. The first argument is the class name, as in the DECLARE_SERIAL macro. The second argument is the name of the base class, from which your class is inherited. The third argument is a version number that can be used to determine whether a file is the correct version for reading into your application. The version number, which must be a positive number, should be incremented each time the serialization method of the class is changed in any way that alters the data being written to or read from a file. A typical usage of the IMPLEMENT_SERIAL macro is provided in Listing 13.3.

LISTING 13.3. INCLUDING THE IMPLEMENT_SERIAL MACRO IN THE CLASS IMPLEMENTATION.

```
 1: // MyClass.cpp: implementation of the CMyClass class.
 2: //
 3: //////////////////////////////////////////////////////////////////////
 4:
 5: #include "stdafx.h"
 6: #include "MyClass.h"
 7:
 8: #ifdef _DEBUG
 9: #undef THIS_FILE
10: static char THIS_FILE[]=__FILE__;
```

```
11: #define new DEBUG_NEW
12: #endif
13:
14: IMPLEMENT_SERIAL (CMyClass, CObject, 1)
15: //////////////////////////////////////////////////////////////////////
16: // Construction/Destruction
17: //////////////////////////////////////////////////////////////////////
18:
19: CMyClass::CMyClass()
20: {
21: }
22:
23: CMyClass::~CMyClass()
24: {
25: }
```

Defining the `Serialize` Function

Along with the two macros, you need to include a `Serialize` function in your class. This function should be declared as a `void` function with a single argument (`CArchive &ar`), public access, and the virtual check box selected—producing the function declaration in Listing 13.2. When you implement the `Serialize` function for your class, you typically use the same approach as that used in the document class, shown in Listing 13.1, where you check to determine whether the file is being written to or read from.

Implementing a Serializable Class

When you begin designing a new application, one of the first things you need to design is how to store the data in the document class that your application will create and operate on. If you are creating a data-oriented application that collects sets of data from the user, much like a contact database application, how are you going to hold that data in the application memory? What if you are building a word processor application—how are you going to hold the document being written in the application memory? Or a spreadsheet? Or a painting program? Or…you get the idea.

Once you determine how you are going to design the data structures on which your application will operate, then you can determine how best to serialize your application and classes. If you are going to hold all data directly in the document class, all you need to worry about is writing the data to and reading the data from the `CArchive` object in the document's `Serialize` function. If you are going to create your own class to hold your application data, you need to add the serialization functionality to your data classes so that they can save and restore themselves.

In the application that you are going to build today, you will create a simple, flat-file database application that illustrates how you can combine a mixture of data types into a single data stream in the application serialization. Your application will display a few fields of data, some of which are variable-length strings, and others that are integer or boolean, and will save and restore them in a single data stream to and from the `CArchive` object.

Creating a Serialized Application

You can create your own classes, which can also be serialized, for use in an SDI or MDI application. In short, any application that works with any sort of data, whether a database or a document, can be serialized. Now you will create a simple, flat-file database application that you will serialize.

A flat-file database is one of the original types of databases. It is a simple file-based database, with the records sequentially appended to the end of the previous record. It has none of the fancy relational functionality that is standard in most databases today. The database that you will build today is closer to an old dBASE or Paradox database, without any indexes, than to databases such as Access or SQL Server.

Creating the Application Shell

To get your application started, create a new AppWizard application. Give your application a name, such as `Serialize`, and click OK to start the AppWizard.

In the AppWizard, select to create a single document style application using the Document/View architecture. You can choose to include support for ActiveX controls in the third AppWizard step, although it's not really necessary for the example that you will build.

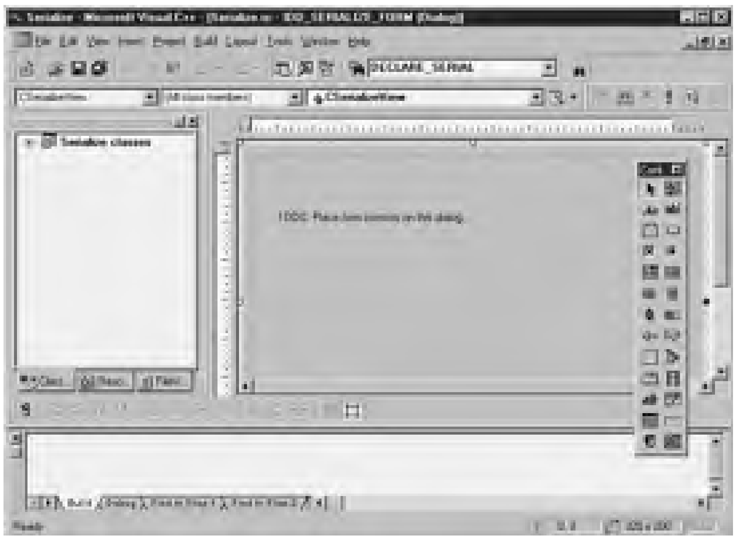
In the fourth step, be sure to specify the file extension for the files that your application will create and read. An example of a file extension that you might want to use is `ser` for serialize or `fdb` for flat-file database.

In the sixth AppWizard step, you need to specify which base class to use for the application view class. For a description of the different base classes available for inheriting the view class from, refer to Day 10 in the section "The Document/View Architecture." For the sample application you are building, because it will be a database application, you'll

find it easiest to use `CFormView` as the base class from which your view class will be inherited. This enables you to use the dialog designer for your application view.

Once you finish making your way through the AppWizard and let the AppWizard create your application shell, you will see a large window canvas in the dialog designer as if you had created a dialog-style application, only without the OK and Cancel buttons, as shown in Figure 13.2.

FIGURE 13.2.
The window designer for an SDI application.



Designing Your Application Window

After you create an SDI or MDI application where the view class is based on the `CFormView` class, you need to design your application view. Designing the view is much like designing the window layout for a dialog window, but you don't need to worry about including any buttons to close the window while either saving or canceling the work done by the user. With an SDI or MDI application, the functionality to save and exit the window is traditionally located on the application menus or on the toolbar. As a result, you need to include only the controls for the function that your application window will perform.

Note

> If you are building dialog-style applications, the AppWizard doesn't provide any serialization code in your application shell. If you need to serialize a dialog-style application, you'll need to add all this code yourself.

For the sample application that you are building today, lay out controls on the window canvas as shown in Figure 13.3 using the control properties listed in Table 13.1.

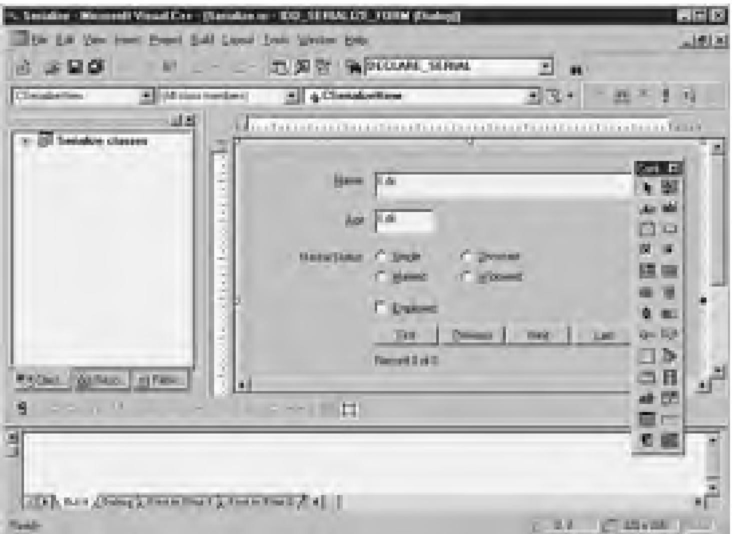


FIGURE 13.3.
The sample application window layout.

TABLE 13.1. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`&Name:`
Edit Box	`ID`	`IDC_ENAME`
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`&Age`
Edit Box	`ID`	`IDC_EAGE`
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`Marital Status:`
Radio Button	`ID`	`IDC_RSINGLE`
	`Caption`	`&Single`
	`Group`	`Checked`
Radio Button	`ID`	`IDC_RMARRIED`
	`Caption`	`&Married`
Radio Button	`ID`	`IDC_RDIVORCED`
	`Caption`	`&Divorced`

Object	*Property*	*Setting*
Radio Button	`ID`	`IDC_RWIDOW`
	`Caption`	`&Widowed`
Check Box	`ID`	`IDC_CBEMPLOYED`
	`Caption`	`&Employed`
Button	`ID`	`IDC_BFIRST`
	`Caption`	`&First`
Button	`ID`	`IDC_BPREV`
	`Caption`	`&Previous`
Button	`ID`	`IDC_BNEXT`
	`Caption`	`Nex&t`
Button	`ID`	`IDC_BLAST`
	`Caption`	`&Last`
Static Text	`ID`	`IDC_SPOSITION`
	`Caption`	`Record 0 of 0`

When you were developing dialog-style applications or windows, you attached variables to the controls on the window in the dialog class. However, with an SDI or MDI application, which class do you create the variables in? Because the `UpdateData` function is a member of the `CWnd` class, and the view class is descended from the `CWnd` class, although the document is not, then the view class is the most logical place to add the variables that you will attach to the controls you placed on the window.

To attach variables to the controls in your sample application, open the Class Wizard and add variables to the controls, specifying that the place to add them is the view class (in this case, `CSerializeView`). For the sample application, add the variables in Table 13.2 to the controls specified.

TABLE 13.2. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
`IDC_CBEMPLOYED`	`m_bEmployed`	`Value`	`BOOL`
`IDC_EAGE`	`m_iAge`	`Value`	`int`
`IDC_ENAME`	`m_sName`	`Value`	`CString`
`IDC_RSINGLE`	`m_iMaritalStatus`	`Value`	`int`
`IDC_SPOSITION`	`m_sPosition`	`Value`	`CString`

If you examine the source code for the view class, you will notice that there is no `OnDraw` function. If you are using the `CFormView` ancestor class for your SDI or MDI application, you don't need to worry about the `OnDraw` function. Instead, you treat the view class very much as you would the dialog class in a dialog window or dialog-style application. The primary difference is that the data that you need to use to populate the controls on the window are not in the view class, but in the document class. As a result, you need to build the interaction between these two classes to pass the data for the controls back and forth.

Creating a Serializable Class

When you create a form-based application, it is assumed that your application will hold multiple records in the form and that the user will be able to scroll through the records to make changes. The user will be able to add additional records or even remove records from the record set. The challenge at this point in building this application is how you represent this set of records, supporting all the necessary functionality.

One approach is to create a class that would encapsulate each record, and then hold these records in an array, much as you did with the drawing application that you created and enhanced over the past few days. This class would need to descend from the `CObject` class and would need to contain variables for all the control variables that you added to the view class, along with methods to read and write all of these variables. Along with adding the methods to set and read all of the variables, you need to make the class serializable by adding the `Serialize` function to the class, as well as the two macros that complete the serialization of the class.

Creating the Basic Class

As you may remember from Day 10, when you want to create a new class, you can select the project in the Class View tab of the workspace pane, right-click the mouse button, and select New Class from the context menu. This opens the New Class dialog.

In the New Class dialog, you specify the type of class, whether it's an MFC class, and generic class, or a form class. To create a class that can contain one record's data, you most likely want to create a generic class. You'll learn more about how to determine which of these types of classes to create on Day 16, "Creating Your Own Classes and Modules." The other things that you need to do are give your class a name and specify the base class from which it will be inherited.

For your sample application, because the form that you created has information about a person, you might want to call your class something like `CPerson`. To be able to hold your class in the object array, you need to give it `CObject` as the base class. Just like on

Day 10, the New Class dialog will claim that it cannot find the header with the base class in it and that you need to add this. Well, it's already included, so you can ignore this message. (On Day 16, you'll learn when you need to pay attention to this message.)

Once you create your new class, you'll need to add the variables for holding the data elements that will be displayed on the screen for the user. Following good object-oriented design, these variables will all be declared as private variables, where they cannot be directly manipulated by other classes. The variable types should match the variable types of the variables that are attached to the window controls in the view class.

With the sample application you are creating, you need to add the variables in Table 13.3.

TABLE 13.3. CLASS VARIABLES FOR THE `CPerson` CLASS.

Name	*Type*
`m_bEmployed`	`BOOL`
`m_iAge`	`int`
`m_sName`	`CString`
`m_iMaritalStatus`	`int`

Adding Methods for Reading and Writing Variables

Once you create your class, you need to provide a means for reading and writing to the variables in the class. One of the easiest ways to provide this functionality is to add inline functions to the class definition. You create a set of inline functions to set each of the variables and then make another set for retrieving the current value of each variable.

An inline function is a short C++ function in which, when the application is being compiled, the function body is copied in place of the function call. As a result, when the compiled application is running, the function code is executed without having to make a context jump to the function and then jump back once the function has completed. This reduces the overhead in the running application, increasing the execution speed slightly, but also makes the resulting executable application slightly larger. The more places the inline function is called, the larger the application will eventually get. For more information on inline functions, consult Appendix A, "C++ Review."

If you want to implement the `Get` and `Set` variable functions for your `CPerson` class in the sample application that you are building, edit the `Person.h` header file, adding the lines in Listing 13.4.

LISTING 13.4. THE `Get` AND `Set` INLINE FUNCTION DECLARATIONS.

```
 1: class CPerson : public CObject
 2: {
 3: public:
 4:     // Functions for setting the variables
 5:     void SetEmployed(BOOL bEmployed) { m_bEmployed = bEmployed;}
 6:     void SetMaritalStat(int iStat) { m_iMaritalStatus = iStat;}
 7:     void SetAge(int iAge) { m_iAge = iAge;}
 8:     void SetName(CString sName) { m_sName = sName;}
 9:     // Functions for getting the current settings of the variables
10:     BOOL GetEmployed() { return m_bEmployed;}
11:     int GetMaritalStatus() { return m_iMaritalStatus;}
12:     int GetAge() {return m_iAge;}
13:     CString GetName() {return m_sName;}
14:     CPerson();
15:     virtual ~CPerson();
16:
17: private:
18:     BOOL m_bEmployed;
19:     int m_iMaritalStatus;
20:     int m_iAge;
21:     CString m_sName;
22: };
```

After you have the methods for setting and retrieving the values of the variables in your custom class, you'll probably want to make sure that the variables are initialized when the class is first created. You can do this in the class constructor by setting each of the variables to a default value. For instance, in your sample application, you add the code in Listing 13.5 to the constructor of the `CPerson` class.

LISTING 13.5. THE `CPerson` CONSTRUCTOR.

```
1: CPerson::CPerson()
2: {
3:     // Initialize the class variables
4:     m_iMaritalStatus = 0;
5:     m_iAge = 0;
6:     m_bEmployed = FALSE;
7:     m_sName = "";
8: }
```

Serializing the Class

After you have your custom class with all variables defined and initialized, you need to make the class serializable. Making your class serializable involves three steps. The first step is adding the `Serialize` function to the class. This function writes the variable values to, and reads them back from, the `CArchive` object using C++ streams. The other two steps consist of adding the `DECLARE_SERIAL` and `IMPLEMENT_SERIAL` macros. Once you add these elements, your custom class will be serializable and ready for your application.

To add the `Serialize` function to your custom class, add a member function through the Class View tab in the workspace pane. Specify the function type as void, the function declaration as `Serialize(CArchive &ar)`, and the access as public and check the Virtual check box. This should add the `Serialize` function and place you in the editor, ready to flesh out the function code.

In the `Serialize` function, the first thing you want to do is to call the ancestor's `Serialize` function. When you call the ancestor's function first, any foundation information that has been saved is restored first, providing the necessary support for your class before the variables in your class are restored. Once you call the ancestor function, you need to determine whether you need to read or write the class variables. You can do this by calling `CArchive`'s `IsStoring` method. This function returns `TRUE` if the archive is being written to and `FALSE` if it's being read from. If the `IsStoring` function returns `TRUE`, you can use C++ I/O streams to write all your class variables to the archive. If the function returns `FALSE`, you can use C++ streams to read from the archive. In both cases, you must be certain to order the variables in the same order for both reading and writing. If you need more information about C++ streams, see Appendix A.

An example of a typical `Serialize` function for your sample custom class is shown in Listing 13.6. Notice that the `CPerson` variables are in the same order when writing to and reading from the archive.

LISTING 13.6. THE `CPerson.Serialize` FUNCTION.

```
 1: void CPerson::Serialize(CArchive &ar)
 2: {
 3:     // Call the ancestor function
 4:     CObject::Serialize(ar);
 5:
 6:     // Are we writing?
 7:     if (ar.IsStoring())
 8:         // Write all of the variables, in order
 9:         ar << m_sName << m_iAge << m_iMaritalStatus << m_bEmployed;
10:     else
11:         // Read all of the variables, in order
12:         ar >> m_sName >> m_iAge >> m_iMaritalStatus >> m_bEmployed;
13: }
```

Once you have the `Serialize` function in place, you need to add the macros to your custom class. The first macro, `DECLARE_SERIAL`, needs to go in the class header and is passed the class name as its only argument.

For example, to add the `DECLARE_SERIAL` macro to the custom `CPerson` class in your sample application, you add the macro just below the start of the class declaration, where it will receive the default access for the class. You specify the class name, `CPerson`, as the only argument to the macro, as in Listing 13.7.

LISTING 13.7. THE SERIALIZED `CPerson` CLASS DECLARATION.

```
 1: class CPerson : public CObject
 2: {
 3:     DECLARE_SERIAL (CPerson)
 4: public:
 5:     // Functions for setting the variables
 6:     void SetEmployed(BOOL bEmployed) { m_bEmployed = bEmployed;}
 7:     void SetMaritalStat(int iStat) { m_iMaritalStatus = iStat;}
 8:     void SetAge(int iAge) { m_iAge = iAge;}
 9:     void SetName(CString sName) { m_sName = sName;}
10:     // Functions for getting the current settings of the variables
11:     BOOL GetEmployed() { return m_bEmployed;}
12:     int GetMaritalStatus() { return m_iMaritalStatus;}
13:     int GetAge() {return m_iAge;}
14:     CString GetName() {return m_sName;}
15:     CPerson();
16:     virtual ~CPerson();
17:
18: private:
19:     BOOL m_bEmployed;
20:     int m_iMaritalStatus;
21:     int m_iAge;
22:     CString m_sName;
23: };
```

Note

The default access permission for functions and variables in C++ classes is public. All functions and variables that are declared before the first access declaration are public by default. You could easily add all of the public class functions and variables in this area of the class declaration, but explicitly declaring the access permission for all functions and variables is better practice—because that way, there is little to no confusion about the visibility of any of the class functions or variables.

Most C++ functions need a semicolon at the end of the line of code. The two serialization macros do not, due to the C preprocessor, which replaces each of the macros with all of the code before compiling the application. It doesn't hurt to place the semicolons there; they are simply ignored.

To complete the serialization of your custom class, you need to add the `IMPLEMENT_SERIAL` macro to the class definition. The best place to add this macro is before the constructor definition in the CPP file containing the class source code. This macro takes three arguments: the custom class name, the base class name, and the version number. If you make any changes to the `Serialize` function, you should increment the version number argument to the `IMPLEMENT_SERIAL` macro. This version number indicates when a file was written using a previous version of the `Serialize` function and thus may not be readable by the current version of the application.

In practice, if you read a file that was written using a previous version of the `Serialize` function in your class, your application will raise an exception, which you can then catch using standard C++ exception-handling techniques. This allows you to add code to your application to recognize and convert files created with earlier versions of your application. For information on C++ exception handling, see Appendix A.

To add the `IMPLEMENT_SERIAL` macro to your sample application, add it into the `Person.cpp` file just before the `CPerson` class constructor. Pass `CPerson` as the first argument (the class name), `CObject` as the second argument (the base class), and `1` as the version number, as in Listing 13.8.

LISTING 13.8. THE `IMPLEMENT_SERIAL` MACRO IN THE `CPerson` CODE.

```
 1: // Person.cpp: implementation of the CPerson class.
 2: //
 3: //////////////////////////////////////////////////////////////////////
 4:
 5: #include "stdafx.h"
 6: #include "Serialize.h"
 7: #include "Person.h"
 8:
 9: #ifdef _DEBUG
10: #undef THIS_FILE
```

continues

LISTING 13.8. CONTINUED

```
11: static char THIS_FILE[]=__FILE__;
12: #define new DEBUG_NEW
13: #endif
14:
15: IMPLEMENT_SERIAL (CPerson, CObject, 1)
16: //////////////////////////////////////////////////////////////////////
17: // Construction/Destruction
18: //////////////////////////////////////////////////////////////////////
19:
20: CPerson::CPerson()
21: {
22:     // Initialize the class variables
23:     m_iMaritalStatus = 0;
24:     m_iAge = 0;
25:     m_bEmployed = FALSE;
26:     m_sName = "";
27: }
```

Building Support in the Document Class

When you build a form-based application, where the form on the window is the primary place for the user to interact with the application, there is an unstated assumption that your application will allow the user to work with a number of records. This means that you need to include support for holding and navigating these records. The support for holding the records can be as simple as adding an object array as a variable to the document class, as you did back on Day 10. This allows you to add additional record objects as needed. The navigation could be a number of functions for retrieving the first, last, next, or previous record objects. Finally, you need informational functionality so that you can determine what record in the set the user is currently editing.

To hold and support this functionality, the document class will probably need two variables, the object array and the current record number in the array. These two variables will provide the necessary support for holding and navigating the record set.

For your example, add the two variables for supporting the record set of `CPerson` objects as listed in Table 13.4. Specify private access for both variables.

TABLE 13.4. DOCUMENT CLASS VARIABLES.

Name	*Type*
`m_iCurPosition`	`int`
`m_oaPeople`	`CObArray`

The other thing that you need to do to the document class to provide support for the record objects is make sure that the document knows about and understands the record object that it will be holding. You do this by including the custom class header file before the header file for the document class is included in the document class source code file. Because the document class needs to trigger actions in the view class, it's a good idea to also include the header file for the view class in the document class.

To include these header files in your sample application, open the source-code file for the document class and add the two `#include` statements as shown in Listing 13.9.

LISTING 13.9. INCLUDING THE CUSTOM AND VIEW CLASSES IN THE DOCUMENT CLASS IMPLEMENTATION.

```
 1: // SerializeDoc.cpp : implementation of the CSerializeDoc class
 2: //
 3:
 4: #include "stdafx.h"
 5: #include "Serialize.h"
 6:
 7: #include "Person.h"
 8: #include "SerializeDoc.h"
 9: #include "SerializeView.h"
10:
11: #ifdef _DEBUG
12: #define new DEBUG_NEW
13: #undef THIS_FILE
14: static char THIS_FILE[] = __FILE__;
15: #endif
16:
17: //////////////////////////////////////////////////////////////////////
18: // CSerializeDoc
```

Adding New Records

Before you can navigate the record set, you need to be able to add new records to the object array. If you add a private function for adding new records, you can add new records to the set dynamically as new records are needed. Because new records should be presenting the user with blank or empty data fields, you don't need to set any of the record variables when adding a new record to the object array, so you can use the default constructor.

Following the same logic that you used to add new line records on Day 10, you should add a new person record to the object array in your document class in today's sample

application. Once you add a new record, you can return a pointer to the new record so that the view class can directly update the variables in the record object.

Once the new record is added, you will want to set the current record position marker to the new record in the array. This way, the current record number can easily be determined by checking the position counter.

If there are any problems in creating the new person record object, let the user know that the application has run out of available memory and delete the allocated object, just as you did on Day 10.

To add this functionality to your sample application, add a new member function to the document class. Specify the type as a pointer to your custom class. If you named your custom class `CPerson`, the function type is `CPerson*`. This function needs no arguments. Give the function a name that reflects what it does, such as `AddNewRecord`. Specify the access for this function as private because it will only be accessed from other functions within the document class. You can edit the resulting function, adding the code in Listing 13.10.

LISTING 13.10. THE `CSerializeDoc.AddNewRecord` FUNCTION.

```
 1: CPerson * CSerializeDoc::AddNewRecord()
 2: {
 3:     // Create a new CPerson object
 4:     CPerson *pPerson = new CPerson();
 5:     try
 6:     {
 7:         // Add the new person to the object array
 8:         m_oaPeople.Add(pPerson);
 9:         // Mark the document as dirty
10:         SetModifiedFlag();
11:         // Set the new position mark
12:         m_iCurPosition = (m_oaPeople.GetSize() - 1);
13:     }
14:     // Did we run into a memory exception?
15:     catch (CMemoryException* perr)
16:     {
17:         // Display a message for the user, giving them the
18:         // bad news
19:         AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
20:         // Did we create a line object?
21:         if (pPerson)
22:         {
23:             // Delete it
24:             delete pPerson;
```

```
25:             pPerson = NULL;
26:         }
27:         // Delete the exception object
28:         perr->Delete();
29:     }
30:     return pPerson;
31: }
```

Getting the Current Position

To aid the user in navigating the record set, it's always helpful to provide a guide about where the user is in the record set. To provide this information, you need to be able to get the current record number and the total number of records from the document to display for the user.

The functions to provide this information are both fairly simple. For the total number of records in the object array, all you need to do is get the size of the array and return that to the caller.

For your sample application, add a new member function to the document class. Specify the function type as `int`, the function name as `GetTotalRecords`, and the access as public. Once you add the function, edit it using the code in Listing 13.11.

LISTING 13.11. THE `CSerializeDoc.GetTotalRecords` FUNCTION.

```
1: int CSerializeDoc::GetTotalRecords()
2: {
3:     // Return the array count
4:     return m_oaPeople.GetSize();
5: }
```

Getting the current record number is almost just as simple. If you are maintaining a position counter in the document class, this variable contains the record number that the user is currently editing. As a result, all you need to do is return the value of this variable to the calling routine. Because the object array begins with position 0, you probably need to add 1 to the current position before returning to display for the user.

To add this function to your sample application, add another new member function to the document class. Specify the type as `int`, the function name as `GetCurRecordNbr`, and the access as public. Edit the function using the code in Listing 13.12.

LISTING 13.12. THE `CSerializeDoc.GetCurRecordNbr` FUNCTION.

```
1: int CSerializeDoc::GetCurRecordNbr()
2: {
3:     // Return the current position
4:     return (m_iCurPosition + 1);
5: }
```

Navigating the Record Set

To make your application really useful, you will need to provide the user with some way of navigating the record set. A base set of functionality for performing this navigation is a set of functions in the document class to get pointers to specific records in the record set. First is a function to get a pointer to the current record. Next are functions to get pointers to the first and last records in the set. Finally, you need functions to get the previous record in the set and the next record in the set. If the user is already editing the last record in the set and attempts to move to the next record, you can automatically add a new record to the set and provide the user with this new, blank record.

To add all this functionality, start with the function to return the current record. This function needs to check the value in the position marker to make sure that the current record is a valid array position. Once it has made sure that the current position is valid, the function can return a pointer to the current record in the array.

To add this function to your sample application, add a new member function to the document class. Specify the function type as `CPerson*` (a pointer to the custom class), the function name as `GetCurRecord`, and the access as public. Edit the function, adding the code in Listing 13.13.

LISTING 13.13. THE `CSerializeDoc.GetCurRecord` FUNCTION.

```
 1: CPerson* CSerializeDoc::GetCurRecord()
 2: {
 3:     // Are we editing a valid record number?
 4:     if (m_iCurPosition >= 0)
 5:         // Yes, return the current record
 6:         return (CPerson*)m_oaPeople[m_iCurPosition];
 7:     else
 8:         // No, return NULL
 9:         return NULL;
10: }
```

The next function you might want to tackle is the function to return the first record in the array. In this function, you need to first check to make sure that the array has records. If

there are records in the array, set the current position marker to 0 and return a pointer to the first record in the array.

To add this function to your sample application, add a new member function to the document class. Specify the function type as `CPerson*` (a pointer to the custom class), the function name as `GetFirstRecord`, and the access as public. Edit the function, adding the code in Listing 13.14.

LISTING 13.14. THE `CSerializeDoc.GetFirstRecord` FUNCTION.

```
 1: CPerson* CSerializeDoc::GetFirstRecord()
 2: {
 3:     // Are there any records in the array?
 4:     if (m_oaPeople.GetSize() > 0)
 5:     {
 6:         // Yes, move to position 0
 7:         m_iCurPosition = 0;
 8:         // Return the record in position 0
 9:         return (CPerson*)m_oaPeople[0];
10:     }
11:     else
12:         // No records, return NULL
13:         return NULL;
14: }
```

For the function to navigate to the next record in the set, you need to increment the current position marker and then check to see if you are past the end of the array. If you are not past the end of the array, you need to return a pointer to the current record in the array. If you are past the end of the array, you need to add a new record to the end of the array.

To add this function to your sample application, add a new member function to the document class. Specify the function type as `CPerson*` (a pointer to the custom class), the function name as `GetNextRecord`, and the access as public. Edit the function, adding the code in Listing 13.15.

LISTING 13.15. THE `CSerializeDoc.GetNextRecord` FUNCTION.

```
 1: CPerson * CSerializeDoc::GetNextRecord()
 2: {
 3:     // After incrementing the position marker, are we
 4:     // past the end of the array?
 5:     if (++m_iCurPosition < m_oaPeople.GetSize())
 6:         // No, return the record at the new current position
```

continues

LISTING 13.15. CONTINUED

```
 7:         return (CPerson*)m_oaPeople[m_iCurPosition];
 8:     else
 9:         // Yes, add a new record
10:         return AddNewRecord();
11: }
```

For the function to navigate to the previous record in the array, you need to make several checks. First, you need to verify that the array has records. If there are records in the array, you need to decrement the current position marker. If the marker is less than zero, you need to set the current position marker to equal zero, pointing at the first record in the array. Once you've made it through all of this, you can return a pointer to the current record in the array.

To add this function to your sample application, add a new member function to the document class. Specify the function type as `CPerson*` (a pointer to the custom class), the function name as `GetPrevRecord`, and the access as public. Edit the function, adding the code in Listing 13.16.

LISTING 13.16. THE `CSerializeDoc.GetPrevRecord` FUNCTION.

```
 1: CPerson * CSerializeDoc::GetPrevRecord()
 2: {
 3:     // Are there any records in the array?
 4:     if (m_oaPeople.GetSize() > 0)
 5:     {
 6:         // Once we decrement the current position,
 7:         // are we below position 0?
 8:         if (--m_iCurPosition < 0)
 9:             // If so, set the record to position 0
10:             m_iCurPosition = 0;
11:         // Return the record at the new current position
12:         return (CPerson*)m_oaPeople[m_iCurPosition];
13:     }
14:     else
15:         // No records, return NULL
16:         return NULL;
17: }
```

For the function that navigates to the last record in the array, you still need to check to make sure that there are records in the array. If the array does have records, you can get the current size of the array and set the current position marker to one less than the number of records in the array. This is actually the last record in the array because the first

record in the array is record 0. Once you set the current position marker, you can return a pointer to the last record in the array.

To add this function to your sample application, add a new member function to the document class. Specify the function type as `CPerson*` (a pointer to the custom class), the function name as `GetLastRecord`, and the access as public. Edit the function, adding the code in Listing 13.17.

LISTING 13.17. THE `CSerializeDoc.GetLastRecord` FUNCTION.

```
 1: CPerson * CSerializeDoc::GetLastRecord()
 2: {
 3:     // Are there any records in the array?
 4:     if (m_oaPeople.GetSize() > 0)
 5:     {
 6:         // Move to the last position in the array
 7:         m_iCurPosition = (m_oaPeople.GetSize() - 1);
 8:         // Return the record in this position
 9:         return (CPerson*)m_oaPeople[m_iCurPosition];
10:     }
11:     else
12:         // No records, return NULL
13:         return NULL;
14: }
```

Serializing the Record Set

When filling in the `Serialize` functionality in the document class, there's little to do other than pass the `CArchive` object to the object array's `Serialize` function, just as you did on Day 10.

When reading data from the archive, the object array will query the `CArchive` object to determine what object type it needs to create and how many it needs to create. The object array will then create each object in the array and call its `Serialize` function, passing the `CArchive` object to each in turn. This enables the objects in the object array to read their own variable values from the `CArchive` object in the same order that they were written.

When writing data to the file archive, the object array will call each object's `Serialize` function in order, passing the `CArchive` object (just as when reading from the archive). This allows each object in the array to write its own variables into the archive as necessary.

For the sample application, edit the document class's `Serialize` function to pass the `CArchive` object to the object array's `Serialize` function, as in Listing 13.18.

Listing 13.18. THE `CSerializeDoc.Serialize` FUNCTION.

```
1: void CSerializeDoc::Serialize(CArchive& ar)
2: {
3:     // Pass the serialization on to the object array
4:     m_oaPeople.Serialize(ar);
5: }
```

Cleaning Up

Now you need to add the code to clean up the document once the document is closed or a new document is opened. This consists of looping through all objects in the object array and deleting each and every one. Once all the objects are deleted, the object array can be reset when you call its `RemoveAll` function.

To implement this functionality in your sample application, add an event-handler function to the document class on the `DeleteContents` event message using the Class Wizard. When editing the function, add the code in Listing 13.19.

LISTING 13.19. THE `CSerializeDoc.DeleteContents` FUNCTION.

```
 1: void CSerializeDoc::DeleteContents()
 2: {
 3:     // TODO: Add your specialized code here and/or call the base class
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the number of lines in the object array
10:     int liCount = m_oaPeople.GetSize();
11:     int liPos;
12:
13:     // Are there any objects in the array?
14:     if (liCount)
15:     {
16:         // Loop through the array, deleting each object
17:         for (liPos = 0; liPos < liCount; liPos++)
18:             delete m_oaPeople[liPos];
19:         // Reset the array
20:         m_oaPeople.RemoveAll();
21:     }
22:
23:     ///////////////////////
24:     // MY CODE ENDS HERE
25:     ///////////////////////
26:
27:     CDocument::DeleteContents();
28: }
```

Opening a New Document

When a new document is started, you need to present the user with an empty form, ready for new information. To make that empty record ready to accept new information, you need to add a new record into the object array, which is otherwise empty. This results in only one record in the object array. Once the new record is added to the array, you must modify the view to show that a new record exists; otherwise, the view will continue to display the last record edited from the previous record set (and the user will probably wonder why your application didn't start a new record set).

To implement this functionality, you will need to edit the `OnNewDocument` function in your document class. This function is already in the document class, so you do not need to add it through the Class Wizard. The first thing that you do in this function is add a new record to the object array. Once the new record is added, you need to get a pointer to the view object. You use the `GetFirstViewPosition` function to get the position of the view object. Using the position returned for the view object, you can use the `GetNextView` function to retrieve a pointer to the view object. Once you have a valid pointer, you can use it to call a function that you will create in the view class to tell the view to refresh the current record information being displayed in the form.

> One thing to keep in mind when writing this code is that you need to cast the pointer to the view as a pointer of the class of your view object. The `GetNextView` function returns a pointer of type `CView`, so you will not be able to call any of your additions to the view class until you cast the pointer to your view class. Casting the pointer tells the compiler that the pointer is really a pointer to your view object class and thus does contain all the functions that you have added. If you don't cast the pointer, the compiler will assume that the view object does not contain any of the functions that you have added and will not allow you to compile your application.

13

Locate the `OnNewDocument` function in the document class source code, and add the code in Listing 13.20. Before you will be able to compile your application, you will need to add the `NewDataSet` function to the view class.

LISTING 13.20. THE `CSerializeDoc.OnNewDocument` FUNCTION.

```
1: BOOL CSerializeDoc::OnNewDocument()
2: {
3:     if (!CDocument::OnNewDocument())
4:         return FALSE;
5:
```

continues

LISTING 13.20. CONTINUED

```
 6:     // TODO: add reinitialization code here
 7:     // (SDI documents will reuse this document)
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // If unable to add a new record, return FALSE
14:     if (!AddNewRecord())
15:         return FALSE;
16:
17:     // Get a pointer to the view
18:     POSITION pos = GetFirstViewPosition();
19:     CSerializeView* pView = (CSerializeView*)GetNextView(pos);
20:     // Tell the view that it's got a new data set
21:     if (pView)
22:         pView->NewDataSet();
23:
24:     ///////////////////////
25:     // MY CODE ENDS HERE
26:     ///////////////////////
27:
28:     return TRUE;
29: }
```

When opening an existing data set, you don't need to add any new records, but you still need to let the view object know that it needs to refresh the record being displayed for the user. As a result, you can add the same code to the `OnOpenDocument` function as you added to the `OnNewDocument`, only leaving out the first part where you added a new record to the object array.

Add an event-handler function to the document class for the `OnOpenDocument` event using the Class Wizard. Once you add the function, edit it adding the code in Listing 13.21.

LISTING 13.21. THE `CSerializeDoc.OnOpenDocument` FUNCTION.

```
1: BOOL CSerializeDoc::OnOpenDocument(LPCTSTR lpszPathName)
2: {
3:     if (!CDocument::OnOpenDocument(lpszPathName))
4:         return FALSE;
5:
6:     // TODO: Add your specialized creation code here
7:
```

```
 8:     ///////////////////////
 9:     // MY CODE STARTS HERE
10:     ///////////////////////
11:
12:     // Get a pointer to the view
13:     POSITION pos = GetFirstViewPosition();
14:     CSerializeView* pView = (CSerializeView*)GetNextView(pos);
15:     // Tell the view that it's got a new data set
16:     if (pView)
17:         pView->NewDataSet();
18:
19:     ///////////////////////
20:     // MY CODE ENDS HERE
21:     ///////////////////////
22:
23:     return TRUE;
24: }
```

Adding Navigating and Editing Support in the View Class

Now that you've added support for the record set to your document class, you need to add the functionality into the view class to navigate, display, and update the records. When you first designed your view class, you placed a number of controls on the window for viewing and editing the various data elements in each record. You also included controls for navigating the record set. Now you need to attach functionality to those controls to perform the record navigation and to update the record with any data changes the user makes.

Because of the amount of direct interaction that the form will have with the record object—reading variable values from the record and writing new values to the record—it makes sense that you want to add a record pointer to the view class as a private variable. For your example, add a new member variable to the view class, specify the type as `CPerson*`, give it a name such as `m_pCurPerson`, and specify the access as private. Next, edit the view source code file and include the header file for the person class, as in Listing 13.22.

13

LISTING 13.22. INCLUDING THE CUSTOM OBJECT HEADER IN THE VIEW CLASS SOURCE CODE.

```
1: // SerializeView.cpp : implementation of the CSerializeView class
2: //
3:
4: #include "stdafx.h"
5: #include "Serialize.h"
6:
```

continues

LISTING 13.22. CONTINUED

```
 7: #include "Person.h"
 8: #include "SerializeDoc.h"
 9: #include "SerializeView.h"
10:
11: #ifdef _DEBUG
12: .
13: .
14: .
```

Displaying the Current Record

The first functionality that you will want to add to the view class is the functionality to display the current record. Because this functionality will be used in several different places within the view class, it makes the most sense to create a separate function to perform this duty. In this function, you get the current values of all the variables in the record object and place those values in the view class variables that are attached to the controls on the window. The other thing that you want to do is get the current record number and the total number of records in the set and display those for the user so that the user knows his or her relative position within the record set.

In your sample application, add a new member function, specify the function type as void, give the function a name that makes sense, such as `PopulateView`, and specify the access as private. In the function, get a pointer to the document object. Once you have a valid pointer to the document, format the position text display with the current record number and the total number of records in the set, using the `GetCurRecordNbr` and `GetTotalRecords` functions that you added to the document class earlier. Next, if you have a valid pointer to a record object, set all the view variables to the values of their respective fields in the record object. Once you set the values of all of the view class variables, update the window with the variable values, as shown in Listing 13.23.

LISTING 13.23. THE `CSerializeView.PopulateView` FUNCTION.

```
 1: void CSerializeView::PopulateView()
 2: {
 3:     // Get a pointer to the current document
 4:     CSerializeDoc* pDoc = GetDocument();
 5:     if (pDoc)
 6:     {
 7:         // Display the current record position in the set
 8:         m_sPosition.Format("Record %d of %d", pDoc->GetCurRecordNbr(),
 9:                 pDoc->GetTotalRecords());
10:     }
```

```
11:     // Do we have a valid record object?
12:     if (m_pCurPerson)
13:     {
14:         // Yes, get all of the record values
15:         m_bEmployed = m_pCurPerson->GetEmployed();
16:         m_iAge = m_pCurPerson->GetAge();
17:         m_sName = m_pCurPerson->GetName();
18:         m_iMaritalStatus = m_pCurPerson->GetMaritalStatus();
19:     }
20:     // Update the display
21:     UpdateData(FALSE);
22: }
```

Navigating the Record Set

If you added navigation buttons to your window when you were designing the form, then adding navigation functionality is a simple matter of adding event-handler functions for each of these navigation buttons and calling the appropriate navigation function in the document. Once the document navigates to the appropriate record in the set, you need to call the function you just created to display the current record. If the document navigation functions are returning pointers to the new current record object, you should capture that pointer before calling the function to display the current record.

To add this functionality to your sample application, add an event handler to the clicked event for the First button using the Class Wizard. In the function, get a pointer to the document object. Once you have a valid pointer to the document, call the document object's `GetFirstRecord` function, capturing the returned object pointer in the view `CPerson` pointer variable. If you receive a valid pointer, call the `PopulateView` function to display the record data, as in Listing 13.24.

LISTING 13.24. THE `CSerializeView.OnBfirst` FUNCTION.

```
 1: void CSerializeView::OnBfirst()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     // Get a pointer to the current document
 6:     CSerializeDoc * pDoc = GetDocument();
 7:     if (pDoc)
 8:     {
 9:         // Get the first record from the document
10:         m_pCurPerson = pDoc->GetFirstRecord();
11:         if (m_pCurPerson)
12:         {
```

continues

LISTING 13.24. CONTINUED

```
13:             // Display the current record
14:             PopulateView();
15:         }
16:     }
17: }
```

For the Last button, perform the same steps as for the First button, but call the document object's `GetLastRecord` function, as in Listing 13.25.

LISTING 13.25. THE `CSerializeView.OnBlast` FUNCTION.

```
 1: void CSerializeView::OnBlast()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     // Get a pointer to the current document
 6:     CSerializeDoc * pDoc = GetDocument();
 7:     if (pDoc)
 8:     {
 9:         // Get the last record from the document
10:         m_pCurPerson = pDoc->GetLastRecord();
11:         if (m_pCurPerson)
12:         {
13:             // Display the current record
14:             PopulateView();
15:         }
16:     }
17: }
```

For the Previous and Next buttons, repeat the same steps again, but call the document object's `GetPrevRecord` and `GetNextRecord` functions. This final step provides your application with all the navigation functionality necessary to move through the record set. Also, because calling the document's `GetNextRecord` on the last record in the set automatically adds a new record to the set, you also have the ability to add new records to the set as needed.

Saving Edits and Changes

When the user enters changes to the data in the controls on the screen, these changes somehow need to make their way into the current record in the document. If you are maintaining a pointer in the view object to the current record object, you can call the record object's various set value functions, passing in the new value, to set the value in the record object.

To implement this in your sample application, add an event handler to the CLICKED event for the Employed check box using the Class Wizard. In the function that you created, first call the UpdateData to copy the values from the form to the view variables. Check to make sure that you have a valid pointer to the current record object, and then call the appropriate Set function on the record object (in this case, the SetEmployed function as in Listing 13.26).

LISTING 13.26. THE CSerializeView.OnCbemployed FUNCTION.

```
 1: void CSerializeView::OnCbemployed()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     // Sync the data in the form with the variables
 6:     UpdateData(TRUE);
 7:     // If we have a valid person object, pass the data changes to it
 8:     if (m_pCurPerson)
 9:         m_pCurPerson->SetEmployed(m_bEmployed);
10: }
```

Repeat these same steps for the other controls, calling the appropriate record object functions. For the Name and Age edit boxes, you add an event handler on the EN_CHANGE event and call the SetName and SetAge functions. For the marital status radio buttons, add an event handler for the BN_CLICKED event and call the same event-handler function for all four radio buttons. In this function, you call the SetMaritalStat function in the record object.

Displaying a New Record Set

The last functionality that you need to add is the function to reset the view whenever a new record set is started or opened so that the user doesn't continue to see the old record set. You will call the event handler for the First button, forcing the view to display the first record in the new set of records.

To implement this functionality in your sample application, add a new member function to the view class. Specify the function type as void, give the function the name that you were calling from the document object (NewDataSet), and specify the access as public (so that it can be called from the document class). In the function, call the First button event handler, as in Listing 13.27.

LISTING 13.27. THE `CSerializeView.NewDataSet` FUNCTION.

```
1: void CSerialize1View::NewDataSet()
2: {
3:     // Display the first record in the set
4:     OnBfirst();
5: }
```

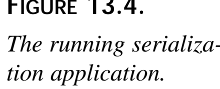
Wrapping Up the Project

Before you can compile and run your application, you need to include the header file for your custom class in the main application source-code file. This file is named the same as your project with the CPP extension. Your custom class header file should be included before the header files for either the document or view classes. For your sample application, you edit the `Serialize.cpp` file, adding line 8 in Listing 13.28.

LISTING 13.28. INCLUDING THE RECORD CLASS HEADER IN THE MAIN SOURCE FILE.

```
 1: // Serialize.cpp : Defines the class behaviors for the application.
 2: //
 3:
 4: #include "stdafx.h"
 5: #include "Serialize.h"
 6:
 7: #include "MainFrm.h"
 8: #include "Person.h"
 9: #include "SerializeDoc.h"
10: #include "SerializeView.h"
11:
12: #ifdef _DEBUG
13: .
14: .
15: .
```

At this point, you can add, edit, save, and restore sets of records with your application. If you compile and run your application, you can create records of yourself and all your family members, your friends, and anyone else you want to include in this application. If you save the record set you create and then reopen the record set the next time that you run your sample application, you should find that the records are restored back to the state that you originally entered them, as in Figure 13.4.



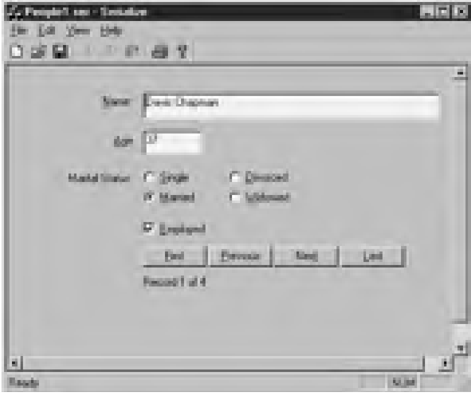


FIGURE 13.4.
The running serialization application.

Summary

Today, you learned quite a bit. You learned how serialization works and what it does. You learned how to make a custom class serializable and why and how to use the two macros that are necessary to serialize a class. You also learned how to design and build a form-based SDI application, maintaining a set of records in a flat-file database for use in the application. You learned how to use serialization to create and maintain the flat-file database and how to construct the functionality in the document and view classes to provide navigating and editing capabilities on these record sets.

Q&A

Q If I make any changes to one of the records in my record set after I save the record set and then I close the application, or open a different set of records, my application doesn't ask if I want to save my changes. How do I get it to ask me? How do I get my application to prompt for saving when data has been changed?

A One function call in the `AddNewRecord` function in the document object is the key to this problem. After adding a new record to the object array, you call the `SetModifiedFlag` function. This function marks the document as "dirty." When you save the record set, the document is automatically set to a "clean" state (unless the application is unable to save the record set for any reason). What you need to do when saving the edits is set the document to the "dirty" state so that the application knows that the document has unsaved changes.

You can fix this by adding some code to each of your data control event handlers. Once you save the new value to the current record, get a pointer to the document object and call the document's `SetModifiedFlag` function, as in Listing 13.29. If you make this same addition to all the data change event handlers, your application will ask you whether to save the changes you made since the last time the record set was saved.

LISTING 13.29. THE MODIFIED `CSerializeView.OnCbemployed` FUNCTION.

```
 1: void CSerializeView::OnCbemployed()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     // Sync the data in the form with the variables
 6:     UpdateData(TRUE);
 7:     // If we have a valid person object, pass the data changes to it
 8:     if (m_pCurPerson)
 9:         m_pCurPerson->SetEmployed(m_bEmployed);
10:     // Get a pointer to the document
11:     CSerializeDoc * pDoc = GetDocument();
12:     if (pDoc)
13:         // Set the modified flag in the document
14:         pDoc->SetModifiedFlag();
15: }
```

Q Why do I need to change the version number in the `IMPLEMENT_SERIAL` macro if I change the `Serialize` function in the record custom class?

A Whether you need to increment the version number depends on the type of change you make. For instance, if you add a calculated field in the record class and you add the code to calculate this new variable from the values you read in the variables from the `CArchive` object, then you don't really need to increment the version number because the variables and order of the variables that you are writing to and reading from the archive did not change. However, if you add a new field to the record class and add the new field into the I/O stream being written to and read from the `CArchive` object, then what you are writing to and reading from the archive will have changed, and you do need to increment the version number. If you don't increment the version number, reading files created using the previous version of your application will result in an "Unexpected file format" message instead of the file being read. Once you increment the version number and you read a file written with the old version number, you get the same message, but you have the option of writing your own code to handle the exception and redirecting the archive to a conversion routine to convert the file to the new file format.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What two macros do you have to add to a class to make it serializable?
2. How can you determine whether the `CArchive` object is reading from or writing to the archive file?
3. What arguments do you need to pass to the `IMPLEMENT_SERIAL` macro?
4. What class do you need to inherit the view class from to be able to use the dialog designer to create a form for the main window in an SDI or MDI application?
5. What type of file does the `CArchive` write to by default?

Exercise

Add a couple of radio buttons to the form to specify the person's sex, as shown in Figure 13.5. Incorporate this change into the `CPerson` class to make the field persistent.

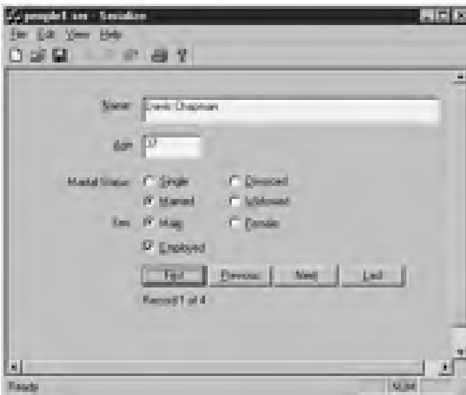


FIGURE 13.5.
The running serialization application with the person's sex.

DAY 14

Retrieving Data from an ODBC Database

A large number of applications use a database. Everything from a personal organizer to a large, corporate personnel system uses a database to store and maintain all the records that the applications use and manipulate. Visual C++ provides you with four different technologies for using and accessing databases in your applications, Data Access Objects (DAO), ODBC, OLE DB, and ActiveX Data Objects (ADO). Today and tomorrow, you'll learn about two of these technologies, how they differ, and how you can use them in your own applications. Today, you will learn

- How the ODBC interface allows you to use a consistent way to access a database.
- How Visual C++ uses the `CRecordset` class to provide access to an ODBC data source.
- How you can create a simple database application using the Wizards in Visual C++.
- How you can add and delete records from an ODBC database in Visual C++.

Database Access and ODBC

Most business applications work with data. They maintain, manipulate, and access records of data that are stored in databases. If you build business applications, odds are that you will need to be able to access a database with your applications. The question is, which database?

There are a number of databases on the market. If you need to create a single-user application that is self-contained on a single computer, you can use any one of numerous PC-based databases, such as Microsoft's Access, FoxPro, or Borland's Paradox. If you are building applications that need to access large, shared databases, you are probably using an SQL-based (Structured Query Language) database such as SQL Server or Oracle. All of these databases provide the same basic functionality, maintaining records of data. Each will allow you to retrieve several records or a single record, depending on your needs. They'll all let you add, update, or delete records as needed. Any of these databases will be able to serve your application's needs, so you should be able to use any database for one application and then switch to another for the next application, based on the needs of the application and which database is most suited for the specific application needs (or your employer's whim).

To be completely honest, there are numerous differences between the various databases that are available today. Each of these databases has specific strengths and weaknesses, making one more suitable for a specific situation than another. However, a discussion of the differences between any of these databases is beyond the scope of this book. For the discussions of databases today and tomorrow, you can assume that all of these databases are functionally equal and interchangeable.

The problem that you will encounter when you switch from one database to another is that each database requires you to use a different interface for accessing the database. Therefore, you have to learn and use a whole new set of programming techniques and functions for each database that you need to work with. This is the problem that the ODBC interface was designed to correct.

The Open Database Connector (ODBC) Interface

Microsoft saw the incompatibility between database interfaces as a problem. Each database had its own application development language that was well integrated with the database but didn't work with any other database. This presented a problem to any developer who needed to use one database for an application and then a different database for

the next application. The developer had to learn the specific development language for each of the databases and couldn't use any languages that she already knew. For programmers to work with any database with the programming language of the developer's choice, they needed a standardized interface that works with every database.

The Open Database Connector (ODBC) interface is implemented as a standard, SQL-based interface that is an integral part of the Windows operating system. Behind this interface are plug-ins for each database that take the ODBC function calls and convert them into calls to the specific interface for that database. The ODBC interface also uses a central set of database connection configurations, with a standardized way of specifying and maintaining them. This setup allows programmers to learn and use a single database interface for all databases. This also allowed programming language vendors to add ODBC support into their languages and development tools to make database access all but transparent.

The `CRecordset` Class

In the Visual C++ development environment, most of the ODBC functionality has been encapsulated into two classes, `CRecordset` and `CDatabase`. The `CDatabase` class contains the database connection information and can be shared across an entire application. The `CRecordset` class encapsulates a set of records from the database. The `CRecordset` class allows you to specify a SQL query to be run, and the `CRecordset` class will run the query and maintain the set of records that are returned by the database. You can modify and update the records in the record set, and your changes will be passed back to the database. You can add or delete records from the record set, and those same actions can be passed back to the database.

Connecting to the Database

Before the `CRecordset` class can perform any other functions, it has to be connected to a database. This is accomplished through the use of the `CDatabase` class. You don't need to create or set the `CDatabase` instance; the first instance of the `CRecordset` class does this for you. When you create an application using the AppWizard and choose to include ODBC database support, the AppWizard includes the database connection information in the first `CRecordset`-derived class that it creates. When this `CRecordset` class is created without being passed a `CDatabase` object, it uses the default connection information, which was added by the AppWizard, to create its own database connection.

Opening and Closing the Record Set

Once the `CRecordset` object is created and connected to the database, you need to open the record set to retrieve the set of records from the database. Do this by calling the `Open` member function of the `CRecordset` object. You can call this function without any

arguments if you want to take the default values for everything, including the SQL statement to be executed.

The first argument to the `Open` function is the record set type. The default value for this, `AFX_DB_USE_DEFAULT_TYPE`, is to open the record set as a snapshot set of records. Table 14.1 lists the four types of record set types. Only two of these record set types are available in the AppWizard when you are specifying the data source.

TABLE 14.1. RECORD SET TYPES.

Type	*Description*
`CRecordset::dynaset`	A set of records that can be refreshed by calling the `Fetch` function so that changes made to the record set by other users can be seen.
`CRecordset::snapshot`	A set of records that cannot be refreshed without closing and then reopening the record set.
`CRecordset::dynamic`	Very similar to the `CRecordset::dynaset` type, but it is not available in many ODBC drivers.
`CRecordset::forwardOnly`	A read-only set of records that can only be scrolled from the first to the last record.

The second argument to the `Open` function is the SQL statement that is to be executed to populate the record set. If a `NULL` is passed for this argument, the default SQL statement that was created by the AppWizard is executed.

The third argument is a set of flags that you can use to specify how the set of records is to be retrieved into the record set. Most of these flags require an in-depth understanding of the ODBC interface so you understand how the flags can and should be used in your applications. Because of this, I'll discuss only a few of these flags in Table 14.2.

TABLE 14.2. RECORD SET OPEN FLAGS.

Flag	*Description*
`CRecordset::none`	The default value for this argument; specifies that no options affect how the record set is opened and used.
`CRecordset::appendOnly`	This flag prevents the user from being able to edit or delete any of the existing records in the record set. The user will only be able to add new records to the set of records. You cannot use this option with the `CRecordset::readOnly` flag.
`CRecordset::readOnly`	This flag specifies that the record set is read-only and no changes can be made by the user. You cannot use this option with the `CRecordset::appendOnly` flag.

Once the user finishes working with the record set, you can call the `Close` function to close the record set and free any resources used by the record set. The `Close` function doesn't take any arguments.

Navigating the Record Set

Once you have a set of records retrieved from the database, you need to be able to navigate the set of records (unless the set has only one record). The `CRecordset` class provides several functions for navigating the record set, allowing you to move the user to any record. Table 14.3 lists the functions that you use to navigate the record set.

TABLE 14.3. RECORD SET NAVIGATION FUNCTIONS.

Function	*Description*
`MoveFirst`	Moves to the first record in the set.
`MoveLast`	Moves to the last record in the set.
`MoveNext`	Moves to the next record in the set.
`MovePrev`	Moves to the previous record in the set.
`Move`	Can be used to move a specific number of records from the current record or from the first record in the set.
`SetAbsolutePosition`	Moves to the specified record in the set.
`IsBOF`	Returns `TRUE` if the current record is the first record in the set.
`IsEOF`	Returns `TRUE` if the current record is the last record in the set.
`GetRecordCount`	Returns the number of records in the set.

Of all of these navigation and informational functions, only two, `Move` and `SetAbsolutePosition`, take any arguments. The `SetAbsolutePosition` function takes a single numeric argument to specify the row number of the record toward which to navigate. If you pass `0`, it navigates to the beginning-of-file (BOF) position, whereas `1` takes you to the first record in the set. You can pass negative numbers to this function to cause it to count backward from the last record in the set. (For example, `-1` takes you to the last record in the set, `-2` to the next-to-last record, and so on.)

The `Move` function takes two arguments. The first argument is the number of rows to move. This can be a positive or negative number; a negative number indicates a backward navigation through the record set. The second argument specifies how you will move through the set of rows. The possible values for the second argument are listed in Table 14.4 with descriptions of how they affect the navigation.

TABLE 14.4. MOVE NAVIGATION TYPES.

Type	*Description*
`SQL_FETCH_RELATIVE`	Moves the specified number of rows from the current row.
`SQL_FETCH_NEXT`	Moves to the next row, ignoring the number of rows specified. The same as calling the `MoveNext` function.
`SQL_FETCH_PRIOR`	Moves to the previous row, ignoring the number of rows specified. The same as calling the `MovePrev` function.
`SQL_FETCH_FIRST`	Moves to the first row, ignoring the number of rows specified. The same as calling the `MoveFirst` function.
`SQL_FETCH_LAST`	Moves to the last row, ignoring the number of rows specified. The same as calling the `MoveLast` function.
`SQL_FETCH_ABSOLUTE`	Moves the specified number of rows from the start of the set of rows. The same as calling the `SetAbsolutePosition` function.

Adding, Deleting, and Updating Records

Navigating a set of records from a database is only part of what you need to be able to do. You also need to be able to add new records to the record set, edit and update existing records, and delete records. These actions are all possible through the various functions that the `CRecordset` class provides. The functions that you will use to provide this functionality to the user are listed in Table 14.5.

TABLE 14.5. RECORD SET EDITING FUNCTIONS.

Function	*Description*
`AddNew`	Adds a new record to the record set.
`Delete`	Deletes the current record from the record set.
`Edit`	Allows the current record to be edited.
`Update`	Saves the current changes to the database.
`Requery`	Reruns the current SQL query to refresh the record set.

None of these functions takes any arguments. However, some of them require following a few specific steps to get them to work correctly.

To add a new record to the database, you can call the `AddNew` function. The next thing that you need to do is set default values in any of the fields that require values, such as the key fields. Next, you must call the `Update` function to add the new record to the database. If you try to navigate to another record before calling the `Update` function, the new

record will be lost. Once you save the new record, you need to call the `Requery` function to refresh the record set so that you can navigate to the new record and let the user edit it. This sequence of function calls typically looks like the following:

```
// Add a new record to the record set
m_pSet.AddNew();
// Set the key field on the new record
m_pSet.m_AddressID = m_lNewID;
// Save the new record to the database
m_pSet.Update();
// Refresh the record set
m_pSet.Requery();
// Move to the new record
m_pSet.MoveLast();
```

When you need to delete the current record, you can simply call the `Delete` function. Once you delete the current record, you need to navigate to another record so the user isn't still looking at the record that was just deleted. Once you delete the current record, there is no current record until you navigate to another one. You do not need to explicitly call the `Update` function because the navigation functions call it for you. This allows you to write the following code to delete the current record:

```
// Delete the current record
m_pSet.Delete();
// Move to the previous record
m_pSet.MovePrev();
```

Finally, to allow the user to edit the current record, you need to call the `Edit` function. This allows you to update the fields in the record with the new values entered by the user or calculated by your application. Once all changes are made to the current record, you need to call the `Update` function to save the changes:

```
// Allow the user to edit the current record
m_pSet.Edit();
// Perform all data exchange, updating the fields in the recordset
.
.
// Save the user's changes to the current record
m_pSet.Update();
```

You might be wondering how you get to the fields in the records to update them. When the AppWizard creates the `CRecordset`-derived class for your application, it adds all the fields in the records that will be in the record set as member variables in order of the record set class. As a result, you can access the member variables in order to access and manipulate the data elements in the database records that are members of the record set.

Creating a Database Application Using ODBC

For the sample application that you will build today, you'll create an SDI application with ODBC database support. The application will retrieve records from an ODBC database, allowing the user to edit and update any of the records. You'll also add functionality to enable the user to add new records to the database and to delete records from the database.

Preparing the Database

Before you can begin building an application that uses a database, you need a database to use with your application. Almost every database that you can purchase for your applications comes with tools for creating a new database. You'll need to use these tools to create your database and then use the ODBC administrator to configure an ODBC data source for your new database.

For the sample application in this chapter, I used Access 95 to create a new database. I used the Access Database Wizard to create the database, choosing the Address Book database template as the database to be created. When the Database Wizard started, I selected the default set of fields for including in the database and selected the option to include sample data, as shown in Figure 14.1. I then accepted the rest of the default settings offered in the Database Wizard.

FIGURE 14.1.
Including sample data in the database.

Once you create the database, you need to configure an ODBC data source to point to the database you just created. To do this, run the ODBC Administrator, which is in the Control Panel on your computer.

Once in the ODBC Administrator, you'll add a new data source. You can do this by clicking the Add button, as shown in Figure 14.2. This opens another dialog, which allows you to select the database driver for the new data source, as shown in Figure 14.3.

For the sample application that you will build today, because the database was created using Access, select the Microsoft Access Driver and click the Finish button.

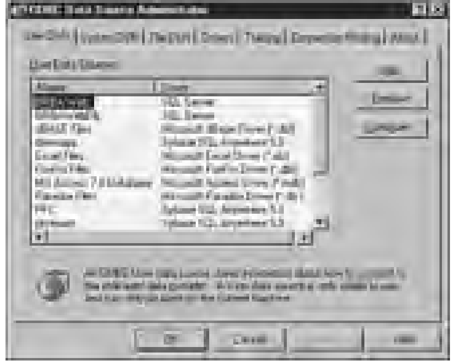


FIGURE 14.2.
The ODBC Data Source Administrator.

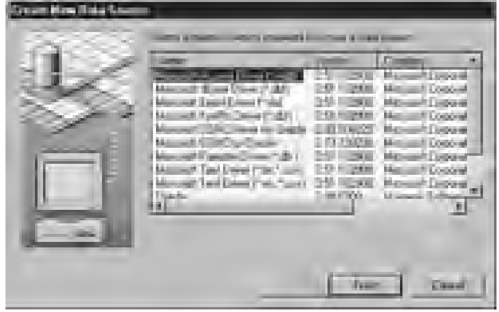


FIGURE 14.3.
The Create New Data Source dialog.

In the ODBC Microsoft Access Setup dialog, shown in Figure 14.4, you'll provide a short, simple name for the data source. Your application will use this name to specify the ODBC data source configuration to use for the database connection, so it should reflect the function that the database will be serving, or it should be similar to the name of the application that will be using this database. For the purposes of the sample application database, name your data source **TYVCDB** (for Teach Yourself Visual C++ Database) and enter a description for the database in the next field.

Once you enter a name and description for the data source, you need to specify where the database is. Click the Select button and then specify the Access database that you created. Once you finish configuring the ODBC data source for your database, click the OK button to add the new data source to the ODBC Administrator. You can click the OK button to finish the task and close the ODBC Administrator because you are now ready to turn your attention to building your application.

FIGURE 14.4.

The ODBC Microsoft Access 97 Setup dialog.

Creating the Application Shell

For the sample application that you will build today, you'll create a standard SDI-style application with database support. First, start a new project, selecting the AppWizard, and give your application a suitable name, such as `DbOdbc`.

On the first AppWizard form, specify that you want to build an SDI application. On the second AppWizard form, specify that you want to include Database view with file support. Click the Data Source button to specify which data source you will use in your application. In the Database Options dialog, specify that you are using an ODBC data source, and select the ODBC configuration from the list that you configured for your Access database, as shown in Figure 14.5. You can set the record set type to either Snapshot or Dynaset.

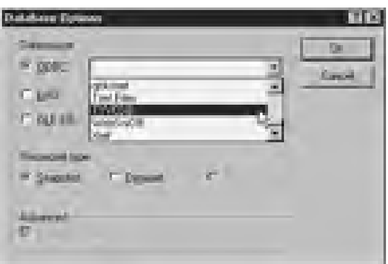


FIGURE 14.5.

The Database Options dialog.

Once you click the OK button, another dialog opens, presenting you with the available tables in the database you selected. Select the Addresses table, as shown in Figure 14.6, and click the OK button to close this dialog and return to the AppWizard.

You can continue through the rest of the AppWizard, accepting all of the default settings. When you reach the final AppWizard step, you'll notice that the AppWizard is going to create an extra class. If you select this class, you'll see that it is derived from the `CRecordset` class, and it is the record set class for your application. You'll also notice

that the view class is derived from the `CRecordView` class, which is a descendent of the `CFormView` class, with some added support for database functionality.

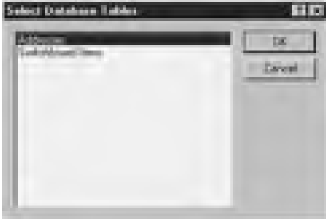


FIGURE 14.6.
The Select Database Tables dialog.

Designing the Main Form

Once you create the application shell, you need to design the main form that will be used for viewing and editing the database records. You can design this form using the standard controls that are part of Visual C++, without adding any special ActiveX controls. For designing the main form in your sample application, lay out the main form as shown in Figure 14.7, and configure the controls with the properties specified in Table 14.6.

Tip

If you want to save a little time when building the example, you can leave out most of the controls and database fields from the application. The key fields that you'll need to include are ID, First and Last Names, Birthdate, and Send Card. If you want to leave out the other fields from the application, that's fine.

TABLE 14.6. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`ID:`
Edit Box	ID	`IDC_EID`
Static Text	ID	`IDC_STATIC`
	Caption	`First Name:`
Edit Box	ID	`IDC_EFNAME`
Static Text	ID	`IDC_STATIC`
	Caption	`Last Name:`
Edit Box	ID	`IDC_ELNAME`

continues

TABLE 14.6. CONTINUED

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`Spouse Name:`
Edit Box	ID	`IDC_ESNAME`
Static Text	ID	`IDC_STATIC`
	Caption	`Address:`
Edit Box	ID	`IDC_EADDR`
	Multiline	`Checked`
Static Text	ID	`IDC_STATIC`
	Caption	`City:`
Edit Box	ID	`IDC_ECITY`
Static Text	ID	`IDC_STATIC`
	Caption	`State:`
Edit Box	ID	`IDC_ESTATE`
Static Text	ID	`IDC_STATIC`
	Caption	`Zip:`
Edit Box	ID	`IDC_EZIP`
Static Text	ID	`IDC_STATIC`
	Caption	`Country:`
Edit Box	ID	`IDC_ECOUNTRY`
Static Text	ID	`IDC_STATIC`
	Caption	`E-Mail:`
Edit Box	ID	`IDC_EEMAIL`
Static Text	ID	`IDC_STATIC`
	Caption	`Home Phone:`
Edit Box	ID	`IDC_EHPHONE`
Static Text	ID	`IDC_STATIC`
	Caption	`Work Phone:`
Edit Box	ID	`IDC_EWPHONE`
Static Text	ID	`IDC_STATIC`
	Caption	`Extension:`
Edit Box	ID	`IDC_EWEXT`
Static Text	ID	`IDC_STATIC`
	Caption	`Fax:`

Object	Property	Setting
Edit Box	ID	`IDC_EFAX`
Static Text	ID	`IDC_STATIC`
	Caption	`Birthdate:`
Edit Box	ID	`IDC_EDOB`
Check Box	ID	`IDC_CBCARD`
	Caption	`Send Card`
Static Text	ID	`IDC_STATIC`
	Caption	`Notes:`
Edit Box	ID	`IDC_ENOTES`
	Multiline	`Checked`

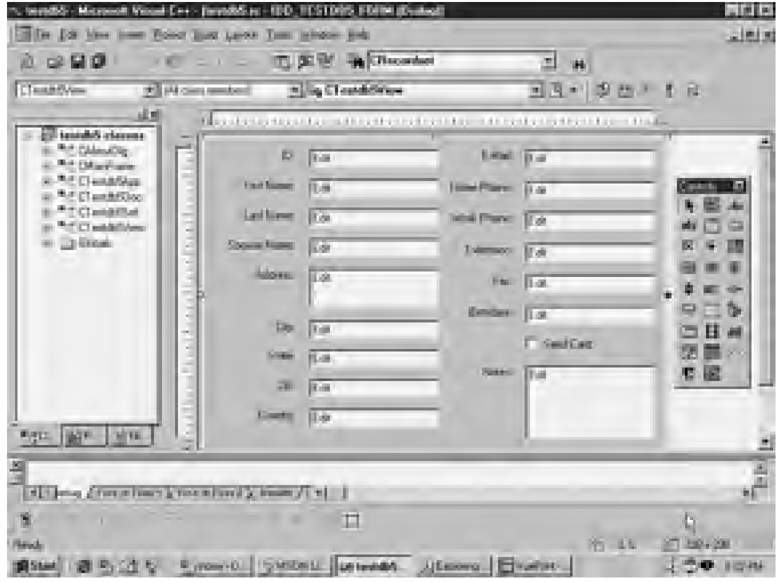


FIGURE 14.7.
The main form design.

Once you have added and configured all the controls on the main form for your application, you're ready to begin associating the controls with database fields. When you click the Member Variables tab of the Class Wizard and select a control to add a variable for, you'll notice that the Add Member Variable dialog has a drop-down combo box where you enter the variable name. If you click the arrow to drop down the list, you'll find that it's filled with the fields in the record set, as shown in Figure 14.8. This enables you to attach the database fields directly to the controls on the form. To attach the database fields to the controls on your application form, add the variables specified in Table 14.7.

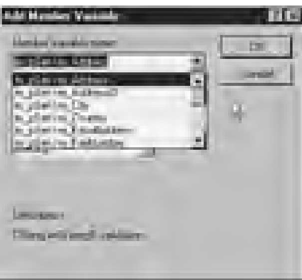


FIGURE 14.8.
The Add Member Variable dialog with record set fields.

TABLE 14.7. CONTROL VARIABLES.

Object	*Name*
IDC_CBCARD	m_pSet->m_SendCard
IDC_EADDR	m_pSet->m_Address
IDC_ECITY	m_pSet->m_City
IDC_ECOUNTRY	m_pSet->m_Country
IDC_EEMAIL	m_pSet->m_EmailAddress
IDC_EFAX	m_pSet->m_FaxNumber
IDC_EFNAME	m_pSet->m_FirstName
IDC_EHPHONE	m_pSet->m_HomePhone
IDC_EID	m_pSet->m_AddressID
IDC_ELNAME	m_pSet->m_LastName
IDC_ENOTES	m_pSet->m_Notes
IDC_ESNAME	m_pSet->m_SpouseName
IDC_ESTATE	m_pSet->m_StateOrProvince
IDC_EWEXT	m_pSet->m_WorkExtension
IDC_EWPHONE	m_pSet->m_WorkPhone
IDC_EZIP	m_pSet->m_PostalCode

You probably noticed when it was time to attach a database field to the birthdate control that the birthday field is missing from the list of database fields. If you look at the record set class in the class view and expand its tree, you'll notice that the birthdate field is included as one of the database fields, but it's not available in the list of available columns for use with the controls. Double-click on the birthdate field in the record set class to view its definition. You'll notice that the `m_Birthdate` variable is declared as a `CTime` variable. This is the reason that it's not available in the list of database fields that can be attached to controls. There isn't a macro or function you can call for exchanging

data between a control and a `CTime` variable. This is also a problem because the `CTime` variable type cannot handle dates before December 31, 1969. To use this database field, you'll need to change its definition from a `CTime` to a `COleDateTime` variable type, as in line 17 in Listing 14.1. Once you change the variable type of this database field, you will be able to attach it to the `IDC_EDOB` control.

LISTING 14.1. THE DATABASE FIELD VARIABLE DECLARATIONS.

```
1:  // Field/Param Data
2:      //{{AFX_FIELD(CTestdb5Set, CRecordset)
3:      long    m_AddressID;
4:      CString    m_FirstName;
5:      CString    m_LastName;
6:      CString    m_SpouseName;
7:      CString    m_Address;
8:      CString    m_City;
9:      CString    m_StateOrProvince;
10:     CString    m_PostalCode;
11:     CString    m_Country;
12:     CString    m_EmailAddress;
13:     CString    m_HomePhone;
14:     CString    m_WorkPhone;
15:     CString    m_WorkExtension;
16:     CString    m_FaxNumber;
17:     COleDateTime    m_Birthdate;
18:     BOOL    m_SendCard;
19:     CString    m_Notes;
20:     //}}AFX_FIELD
```

Note

Normally, you do not want to edit the portions of code in your applications that are created and maintained by the various wizards. The change I outline here is one of the few exceptions to this rule. This obstacle could possibly be considered a bug in the Visual C++ AppWizard, although it's technically not a bug. You can convert the date/time database field to several variable types when creating a class variable to represent that field. `CTime` is one of these variable types; `COleDateTime` is another. Because these are both equally valid choices, and the functions that populate this variable can work with either, making this change is possible without dire consequences.

Once you make the change to the variable type for the m`_Birthdate` variable in the record set class (`CDbOdbcSet`), and attach this database field to the Birthdate control on the form, you might think that you are ready to compile and run your application. Unfortunately, your application will not compile. You'll get a compiler error stating that

the DDX_FieldText cannot convert the COleDateTime variable type. What you need to do is add the code to perform this conversion yourself. Return to the Class Wizard and delete the variable that you added to the IDC_EDOB control. Add a new variable to this control. Specify that the variable is type COleDateTime, and give the variable a name such as m_oledtDOB. Pull up the DoDataExchange function in the view class, CDbOdbcView, into the editor, and add lines 4 through 6 and lines 26 through 28 to the function, as shown in Listing 14.2.

LISTING 14.2. THE CDbOdbcView DoDataExchange FUNCTION.

```
1:  void CDbOdbcView::DoDataExchange(CDataExchange* pDX)
2:  {
3:      CRecordView::DoDataExchange(pDX);
4:      // Copy the DOB from the record set to the view variable
5:      if (pDX->m_bSaveAndValidate == FALSE)
6:          m_oledtDOB = m_pSet->m_Birthdate;
7:      //{{AFX_DATA_MAP(CTestdb5View)
8:      DDX_FieldText(pDX, IDC_EID, m_pSet->m_AddressID, m_pSet);
9:      DDX_FieldText(pDX, IDC_EFNAME, m_pSet->m_FirstName, m_pSet);
10:     DDX_FieldText(pDX, IDC_ELNAME, m_pSet->m_LastName, m_pSet);
11:     DDX_FieldText(pDX, IDC_ESNAME, m_pSet->m_SpouseName, m_pSet);
12:     DDX_FieldText(pDX, IDC_ESTATE, m_pSet->m_StateOrProvince, m_pSet);
13:     DDX_FieldText(pDX, IDC_ECITY, m_pSet->m_City, m_pSet);
14:     DDX_FieldText(pDX, IDC_EADDR, m_pSet->m_Address, m_pSet);
15:     DDX_FieldCheck(pDX, IDC_CBCARD, m_pSet->m_SendCard, m_pSet);
16:     DDX_FieldText(pDX, IDC_ECOUNTRY, m_pSet->m_Country, m_pSet);
17:     DDX_FieldText(pDX, IDC_EEMAIL, m_pSet->m_EmailAddress, m_pSet);
18:     DDX_FieldText(pDX, IDC_EFAX, m_pSet->m_FaxNumber, m_pSet);
19:     DDX_FieldText(pDX, IDC_EHPHONE, m_pSet->m_HomePhone, m_pSet);
20:     DDX_FieldText(pDX, IDC_ENOTES, m_pSet->m_Notes, m_pSet);
21:     DDX_FieldText(pDX, IDC_EWEXT, m_pSet->m_WorkExtension, m_pSet);
22:     DDX_FieldText(pDX, IDC_EWPHONE, m_pSet->m_WorkPhone, m_pSet);
23:     DDX_FieldText(pDX, IDC_EZIP, m_pSet->m_PostalCode, m_pSet);
24:     DDX_Text(pDX, IDC_EDOB, m_oledtDOB);
25:     //}}AFX_DATA_MAP
26:     // Copy the DOB variable back from the view variable to the record
        ➥set
27:     if (pDX->m_bSaveAndValidate == TRUE)
28:         m_pSet->m_Birthdate = m_oledtDOB;
29: }
```

In addition to the above change, you have to remove the initialization of the m_Birthdate variable in the set class. This is also code that was added by the AppWizard, and once again you have to break the rules by modifying the code that you are never supposed to touch. To make this change, you can take the simple approach by

commenting out the initialization of this variable in the set class constructor, in line 19 of Listing 14.3.

LISTING 14.3. THE `CDbOdbcSet` CONSTRUCTOR.

```
1: CDbOdbcSet::CDbOdbcSet(CDatabase* pdb)
2:      : CRecordset(pdb)
3:  {
4:      //{{AFX_FIELD_INIT(CTestdb5Set)
5:      m_AddressID = 0;
6:      m_FirstName = _T("");
7:      m_LastName = _T("");
8:      m_SpouseName = _T("");
9:      m_Address = _T("");
10:     m_City = _T("");
11:     m_StateOrProvince = _T("");
12:     m_PostalCode = _T("");
13:     m_Country = _T("");
14:     m_EmailAddress = _T("");
15:     m_HomePhone = _T("");
16:     m_WorkPhone = _T("");
17:     m_WorkExtension = _T("");
18:     m_FaxNumber = _T("");
19:     //m_Birthdate = 0;
20:     m_SendCard = FALSE;
21:     m_Notes = _T("");
22:     m_nFields = 17;
23:     //}}AFX_FIELD_INIT
24:     m_nDefaultType = dynaset;
25: }
```

Now compile and run your application once again. You'll find that you have a fully functioning database application that retrieves a set of records from the database and allows you to scroll through them and make changes to the data, as shown in Figure 14.9.

Adding New Records

You've already created a fully functioning database application without writing a single line of code. However, a few functions are missing. Most database applications let the user add new records to the database. To add a new record to the database, you'll want to figure out what the next ID number should be, so you'll scroll to the last record in the set to get the ID and then increment it by one. Next, you'll call the `AddNew` function to add a new record, set the ID field to the new ID you calculated, and then call the `Update` function to save the new record. Finally, you'll call the `Requery` function to refresh the set of records and then scroll to the last record in the set to let the user enter data into the new record.

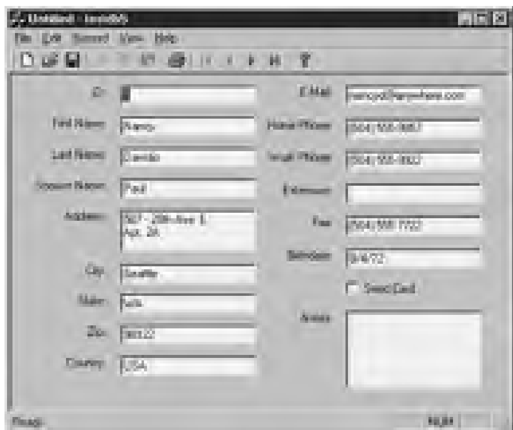


FIGURE 14.9. *The running application.*

Because the ID field in the database in defined as an `AutoIncrement` field, you do not normally specify your own ID for the field. However, because the record set is creating a new record with the ID field, you need to assign a valid ID to the record or you won't be able to add it to the database. The method used in this application will not work with any multiuser database because each person would generate the same IDs for new records. In this situation, a centralized method for generating new IDs, such as a counter field in the database, is a better solution. The other option is to create a SQL statement to insert a new record into the database that was missing the ID field. This allows the auto-increment functionality to work correctly.

To add this functionality to your application, start by adding a function to your record set class to determine the next ID number to be used. Add a member function to the record set class, `CDbOdbcSet`. Specify the function type as long, the function declaration as `GetMaxID`, and the access as public. Edit the function, adding the code in Listing 14.4.

LISTING 14.4. THE `CDbOdbcSet GetMaxID` FUNCTION.

```
1: long CDbOdbcSet::GetMaxID()
2: {
3:     // Move to the last record
4:     MoveLast();
5:     // return the ID of this record
6:     return m_AddressID;
7: }
```

Next, you'll need a menu entry that the user can select to add a new record to the database. Add a new menu entry to the Record menu. Configure the new menu entry with the properties in Table 14.8.

TABLE 14.8. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	ID	`IDM_RECORD_NEW`
	Caption	`N&ew Record`
	Prompt	`Add a new record\nNew Record`

Using the Class Wizard, add an event-handler function for the `COMMAND` event message for this menu to the view class, `CDbOdbcView`. Edit this function, adding the code in Listing 14.5.

LISTING 14.5. THE `CDbOdbcView` `OnRecordNew` FUNCTION.

```
1:  void CDbOdbcView::OnRecordNew()
2:  {
3:      // TODO: Add your command handler code here
4:      // Get a pointer to the record set
5:      CRecordset* pSet = OnGetRecordset();
6:      // Make sure that any changes to the current record
7:      // have been saved
8:      if (pSet->CanUpdate() && !pSet->IsDeleted())
9:      {
10:         pSet->Edit();
11:         if (!UpdateData())
12:             return;
13:
14:         pSet->Update();
15:     }
16:     // Get the ID for the new record
17:     long m_lNewID = m_pSet->GetMaxID() + 1;
18:     // Add the new record
19:     m_pSet->AddNew();
20:     // Set the ID in the new record
21:     m_pSet->m_AddressID = m_lNewID;
22:     // Save the new record
23:     m_pSet->Update();
24:     // Refresh the record set
25:     m_pSet->Requery();
26:     // Move to the new record
27:     m_pSet->MoveLast();
28:     // Update the form
29:     UpdateData(FALSE);
30: }
```

Add a new toolbar button for the New Record menu, and then compile and run your application. You should be able to add new records to the database, entering the data you want into the records.

Deleting Records

The only functionality remaining is the ability to delete the current record from the database. You'll need to add another menu entry to trigger this action. Once the action is triggered, you'll verify that the user really does want to delete the current record and then call the `Delete` function to remove the record. Once the record has been deleted, you'll call the `MovePrev` function to navigate to the previous record in the set.

To add this functionality to your application, you'll need a menu entry that the user can select to delete the current record from the database. Add a new menu entry to the Record menu. Configure the new menu entry with the properties in Table 14.9.

TABLE 14.9. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	ID	`IDM_RECORD_DELETE`
	Caption	`&Delete Record`
	Prompt	`Delete the current record\nDelete Record`

Using the Class Wizard, add an event-handler function for the `COMMAND` event message for this menu to the view class, `CDbOdbcView`. Edit this function, adding the code in Listing 14.6.

LISTING 14.6. THE `CDbOdbcView` `OnRecordDelete` FUNCTION.

```
1:  void CTestdb5View::OnRecordDelete()
2:  {
3:      // TODO: Add your command handler code here
4:      // Make sure the user wants to delete this record
5:      if (MessageBox("Are you sure you want to delete this record?",
6:              "Delete this record?", MB_YESNO ¦ MB_ICONQUESTION) ==
                ➥IDYES)
7:      {
8:          // Delete the record
9:          m_pSet->Delete();
10:         // Move to the previous record
11:         m_pSet->MovePrev();
12:         // Update the form
13:         UpdateData(FALSE);
14:     }
15: }
```

Add another button to the toolbar and associate it with the `IDM_RECORD_DELETE` menu ID so that the user can delete the current record without having to go to the menu. If you compile and run your application at this point, you'll have a full-function database application in which you can add, edit, and delete records, as shown in Figure 14.10.

FIGURE 14.10.
The completed application.

Summary

Today, you learned how you can use the ODBC interface to build database applications that can be easily run against any database you might need to use. You saw how the `CRecordset` class provides you with a substantial amount of functionality so that you can provide database functionality in your applications. You also saw how the AppWizard provides you with a large amount of database functionality without your typing a single line of code.

Tomorrow, you will learn about Microsoft's newest database access technology, ActiveX Data Objects, and how this can be combined with the ODBC interface to make your database access even easier.

Q&A

Q Why would I want to use the ODBC interface instead of the Data Access Objects?

A The Data Access Objects (DAO) use the Microsoft Jet database engine to perform all of the database access. This adds at least a megabyte of overhead to your application, and if you're using a SQL-based database, the database is already doing all of the work that the Jet engine is doing for you. What's more, the Jet database

engine uses the ODBC interface to access any SQL-based databases. As a result, unless you are using PC-based databases, such as Access, FoxPro, or Paradox, you get better performance from going directly to the ODBC interface yourself.

Q How can I add different record sets in an MDI application?

A You can add additional `CRecordset`-derived classes through the New Class Wizard in an MDI application project. You need to specify that the new class is an MFC class and that its base class is the `CRecordset` class. The New Class Wizard will have you specify the data source, just as the AppWizard had you do when creating the shell for today's application. Once you create the record set class, you can create a new view class the same way, specifying the base class as `CRecordView`. Once you click the OK button, the New Class Wizard asks you to specify which of the record set classes to use with the new record view class.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What does ODBC stand for?
2. What functions can you use to navigate the record set in a `CRecordset` object?
3. What view class should you use with an ODBC application?
4. What sequence of functions do you need to call to add a new record to a record set?
5. What function do you need to call before the fields in the `CRecordset` object can be updated with any changes?

Exercise

Add a menu entry and dialog to let the user indicate the record number to move to, and then move to that record.

WEEK 2

In Review

Now that you've finished the second week, you should be getting very comfortable working with Visual C++. You should be beginning to understand how you can use the MFC class hierarchy to provide a substantial amount of existing functionality in your applications. You should also be starting to understand how much supporting infrastructure your applications start with when you use the Visual C++ wizards to construct as much of your application as you can.

This is a good time to take a little break and try some of the things that you've learned on your own. Build an MDI application, using a custom document type that you've come up with yourself. See how you can save and restore the document, as well as maintain it. Practicing on your own is key to cementing your understanding of what you've learned in this book. This will help you identify any areas that you might need to go back and read again, as well as those areas where you feel comfortable enough to not review.

By this time, you should have a good understanding of the Document/View architecture and how it can be used to maintain the separation of the data from the representation of the data that is displayed for the user. You've used this model for both Single Document Interface (SDI) and Multiple Document Interface (MDI) style applications, and you've used it for reading and writing files to the disk drive. This model is one of the main building blocks of MFC applications built with Visual C++. You should know where to place any initialization information for a new set of data and where to clean up when closing a set of data.

You should also have a good understanding of how the SDI and MDI application styles are alike and how they differ from each other and from the dialog application style. You should have a good idea of when an application you are building should use one of these styles and when it should use a different style. You should be able to create your own SDI and MDI applications, as you need to, without any significant problems. If you've got any questions about either of these areas, you might want to take another look at Days 10 and 11 to review how the Document/View architecture works in both SDI and MDI style applications.

You should understand how, in SDI and MDI style applications, you can save and restore complex data structures in files on the system hard drive. You should be able to create mixed-type objects that you create and maintain in the document object in your applications, be able to use the `Serialize` function with the `CArchive` object to write the objects to a file, and then be able to restore the objects at a later time. If you are having any trouble understanding how this works or are running into any problems trying to implement this functionality in your own applications, review Day 13.

Along with reading and writing files, you also have learned how you can design and build toolbars for use in your SDI and MDI applications. At this point, you should be completely comfortable with designing and creating your own toolbars and using them in your applications. You should understand the importance of matching the toolbar button ID to the ID of the menu for which the toolbar will be used as a substitute. You should also have a basic understanding of creating and using your own customized status bar elements in SDI and MDI applications. You should understand how you can use the `UPDATE_COMMAND_UI` event message to evaluate and alter the status of menu, toolbar, and status bar elements, relieving you of all the work of setting each of these elements, and how to maintain their appearance and status yourself. If you aren't clear on how you can do any of these things, you might want to go back over Day 12 one more time.

You've seen how you can build a simple database application, pulling data from a database through the ODBC interface. You should have a basic understanding of how you can build database applications using this approach, how to maintain the data, how to add new records, and how to delete records. You should know how all the database interaction is directed through the record set class and how you can directly control the data through this object. If you're not sure of some of this, you might want to look back at Day 14 for a quick refresher.

You learned how easy it is to add ActiveX controls to your projects and how Visual C++ builds C++ classes around the control, enabling you to interact with the control as if it were just another C++ object. You should have a good grasp of how to add any ActiveX control (armed with the documentation for the control) to your application and interact

with it in a seamless manner. You should be able to declare a variable for the control, set the control's properties, call its methods, and react to its events just as if it were a standard part of the Visual C++ development environment. If you aren't sure how you can do some of this, you might want to go back and reread Day 9.

Finally, you started this week by learning how to draw graphics on the windows of your applications. You learned how to draw lines, circles, and squares, using a variety of pens and brushes. You even learned how you can make a customized brush from a bitmap. You learned how you can load a bitmap image from a file and display it for the user to see. But most importantly, you learned about the device context and how it is used to draw all these features on the windows of your applications. You should be able to use these and other figure drawing device context methods to draw any image you might want to draw on the window for the user to see and interact with. If you are unsure about how you can do this, you probably want to look back at Day 8 once more.

By this time, you have built up quite a set of programming skills with Visual C++. You are probably ready to tackle most of the smaller programming tasks you might encounter—and maybe even a few not-so-small tasks. At this point, you are well on your way to becoming an accomplished Visual C++ programmer. That said—now is not the time to stop because there's still more to be learned. There's only one more week to go, so tallyho!

WEEK 3

At a Glance

For the third and final week, you'll be learning about several of the more advanced aspects of building applications with Visual C++. Some of these topics you'll use more than others, but if you do much work with Visual C++, odds are that you'll work with most, if not all, of these areas before long.

You'll begin the week by picking up where you left off the previous week with building database applications. On Day 15, you'll learn about Microsoft's latest database access technology, ActiveX Data Objects (ADO), and how you can incorporate it into your Visual C++ applications to provide database access to your application's users. You'll learn how using ADO is similar to and different from building database applications using ODBC.

On Day 16, you'll learn how to create your own custom classes and how to approach the design of these classes. You'll also learn how to build your functionality into library modules that you can give to other Visual C++ programmers for use in their applications.

On Day 17, you'll learn a different means of allowing other programmers to use your code by building DLLs. You'll learn how to build two different types of DLLs: those that can be used only by other Visual C++ applications and those that can be used by applications built with any other Windows development language or tool.

On Day 18, you'll learn how you can enable your applications to work on two or more separate tasks at the same time. You'll learn how to trigger some background processing whenever your application is sitting idle and how to spin off independent threads that continue to work even when your application is busy.

On Day 19, you'll learn how to build your own ActiveX controls that can be used in other applications or even in Web pages. You'll see how you can define the properties and methods for your control and how you can trigger events in the containing application from your control.

On Day 20, you'll learn how Internet applications communicate with each other using the Winsock interface. You'll learn how you can use this same interface to enable your applications to communicate over a network or even on the same machine.

Finally, on Day 21, you'll see how easy it is to incorporate the Microsoft Internet Explorer Web browser into your own Visual C++ application. You'll learn how you can control the Web browser, specifying what Web pages for it to display, and how you can provide the user with information about what the browser is doing.

When you finish this final week, you'll be knowledgeable about most areas of Visual C++ programming. Although there will still be areas and technologies that require more in-depth study for you to master, you'll know and understand what those areas are all about. You'll be prepared to dive head first into all areas of Visual C++ programming, and by then you might already have identified some areas that you want to learn more about than can be covered in this book.

You have only one week left to go, so go ahead and dive in and get going.

DAY 15

Updating and Adding Database Records Through ADO

Now that you've gotten your feet wet with an ODBC database application, one of the oldest Microsoft database access technologies, it's time to turn your attention to the newest Microsoft database access technology, ActiveX Data Objects (ADO). Designed for use with all of Microsoft's programming and scripting technologies, ADO presents the Visual C++ programmer with new challenges in database programming, while still keeping the functionality familiar. Today, you will learn

- How ADO works and how it uses the OLE DB technology for providing simple database access.
- How you can build a simple ADO application in a couple of minutes using ActiveX controls.

- How you can build a complete database application using regular forms controls.
- How you can use special ADO macros to build a custom record set class for use in your database applications.

This chapter works with some features that may not be included in all versions of Visual C++. Although ADO is an important new area of programming with Microsoft data access technologies, this chapter discusses some things that you may not have the ability to do with your version of Visual C++.

What Is ADO?

A couple years ago, Microsoft designed a new data access technology called OLE DB. This data access technology was intended to be much more than simply a way of getting data into and out of databases. This technology was intended to be the means of accessing data, regardless of where that data may be located. Through the OLE DB technology, you could access mail messages, spreadsheets, files, and so on. Anything that might have data could possibly be accessed through the OLE DB technology. This was one of the first technologies to be produced from the research and development of the object-oriented file system at the heart of what Microsoft has been calling "Cairo" for the past few years.

Many of the technologies bundled under the product name of Cairo will be released some time next year in the Windows NT 5.0 operating system.

As you can imagine, with the range of functionality that OLE DB must have to access data in all of those different sources, it might be quite complex to work with this technology. Well, it is. This is where ActiveX Data Objects come into play. ADO was designed as another layer on top of OLE DB, specifically for providing database access.

One of the goals in designing ADO was to create a control that could be used to provide data access and control in Web pages, caching the data records on the client. Part of the reason for this goal was to allow a Web browser user to access an entire set of data records, without having to pull down each individual record, one at a time, to navigate and make changes to the records. Because of this capability with ADO, the ADO control is distributed with Microsoft's Internet Explorer Web browser (version 4.0 and above).

ADO Objects

To make ADO as easily usable in scripting languages such as VBScript as it is in programming environments such as Visual Basic, Microsoft tried to keep the number of objects to a minimum. As a result, you have a small number of basic objects:

- `Connection`
- `Error`
- `Command`
- `Parameter`
- `Recordset`
- `Field`

Along with these objects, you have collection objects for containing collections of `Error`, `Parameter`, and `Field` objects.

The `Connection` Object

The `Connection` object is used for establishing and maintaining a connection to a database. This object is configured with the connection information, including database location, user ID, and password, before opening the connection. Once all of this information is appropriately configured, the connection object should have its `Open` method called to open the connection. Once the `Connection` object goes out of scope, the connection is automatically closed. If you want more control over closing and opening the database connection, you can call the `Connection` object's `Close` method to close the connection.

The `Connection` object is also the object through which any high-level connection functionality is controlled. This includes all transaction control, through the `Connection` object's `BeginTrans`, `CommitTrans`, and `RollbackTrans` methods.

The `Error` Object

Whenever a database error occurs, the error information from the database is placed into an ADO `Error` object. The error information in the error object is the database error information, not ADO error information. Whenever you encounter an error and need to look up the error information to determine what went wrong, you'll need to examine the database error codes and descriptions, not the ADO error codes.

The `Command` Object

The `Command` object is used to execute commands in the database. You can use this object to run SQL statements or call stored procedures (SQL functions that are stored in the

database). Any time that a command returns rows of data, you need to attach the `Command` object to a `Recordset` object for the returned data to be stored in.

When you call a stored procedure, as with functions in any other programming language, you'll often need to pass parameters to the stored procedure. To pass these parameters, you'll attach a series of `Parameter` objects to the `Command` object. Each of the `Parameter` objects will have the name of the parameter that it holds the value for, along with the value that should be passed to the database for that particular parameter.

The `Parameter` Object

The `Parameter` object is used for passing variables and for calling stored procedures or parameterized queries. These are attached to a `Command` object for use in calling the command that has been programmed into the `Command` object.

The `Recordset` Object

The `Recordset` object contains a set of records from the database. The set of records is the result of a command being sent to the database that results in a set of records being returned. You can navigate through the `Recordset`, much like you do with the `Recordset` objects for other database access technologies. You can also access the fields in each record in the `Recordset` through the `Field` objects that are associated with the `Recordset`. You can update the records in the `Recordset`, and then use the `Recordset` to update the database. You can also insert new records into the `Recordset`, or delete records and have those changes made in the database.

The `Field` Object

The `Field` object represents a single column in the `Recordset`. Each `Field` object contains the column name, data value, and how the data value should be represented. Because ADO was designed to be used in Microsoft's scripting languages, and the only data type available in these scripting languages is the `Variant` data type, the `Field` objects always contain a `Variant` data value. The data value is automatically converted to the correct data type when updating to the database. As the programmer working with the ADO objects, you will have to convert the value from a `Variant` to whatever data type you need it to be, as well as convert it back to a `Variant` when updating the value.

Using the ADO ActiveX Control

There are two different ways in which you can use the ADO control in your Visual C++ applications. The simple way to incorporate ADO into your application is through the use of ActiveX controls. You can add the ADO data control to your Visual C++ project, just like any other ActiveX control, as shown in Figure 15.1.

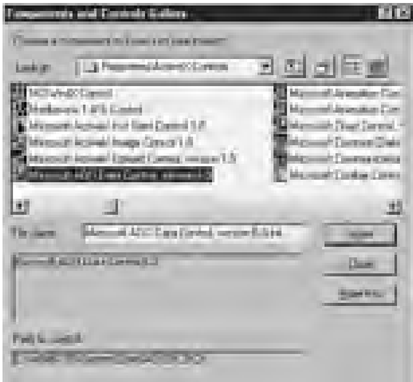


FIGURE 15.1.
Adding the ADO ActiveX control to a project.

Once you add the ADO control to your project, and place it on a window, you'll need to specify the data connection in the control properties, as shown in Figure 15.2. You'll also need to specify the source for the records that will be retrieved by the control, as shown in Figure 15.3.

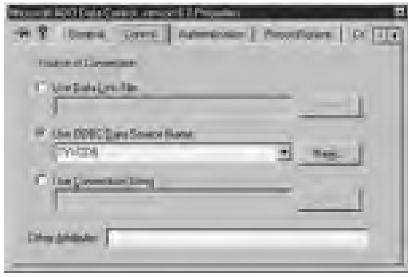


FIGURE 15.2.
Specifying the database connection.

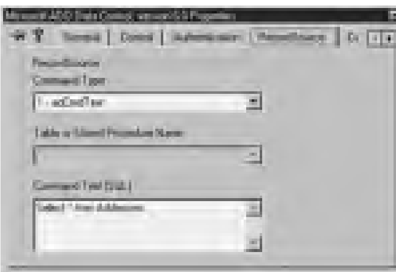


FIGURE 15.3.
Specifying the record source.

To use the ADO control efficiently, you'll also want to use data-bound controls that are ADO-enabled, such as the Microsoft DataGrid control. When you add these controls to the window with the ADO control, you'll specify the ADO control as the data source

for the control, as shown in Figure 15.4. If the control is designed to only provide access to a single field in a record set, you'll also need to specify which field is to be used for the control.

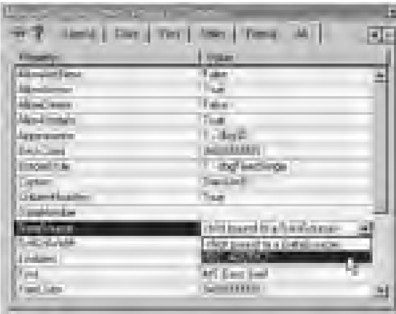


FIGURE 15.4.
Specifying the data source.

Once you add all these controls to the window and configure them, you can run your application and have full database access through ADO without having written a single line of code, as shown in Figure 15.5.

FIGURE 15.5.
A running ADO control database application.

This is such a simple way to build database applications: Just place controls on a window and configure the properties to tell it where to get the data. What's the downside of building ADO applications this way? First, using this approach involves a lot of unnecessary overhead in building ADO applications. For each SQL query or table that you want to pull in a separate record set, you have to add a separate ADO control. Each of these ADO controls will establish a separate connection to the database, which could cause problems with databases that have a limited number of connections available (not to

mention the additional overhead on the application). Finally, not all data-bound controls are ADO enabled. ADO is such a new technology that there are few controls that you can use with it at this time. You may find some controls that allow you to retrieve and display data for the user, but do not allow the user to change and edit the data. Others may not even provide you with that much functionality.

Importing the ADO DLL

If you look around in the MFC class hierarchy, you'll find that there are no classes for use with ADO. If you don't want to use the controls approach, then what are your options? Do you have to create the classes yourself? No, Microsoft has provided other means for you to create and use classes for each of the objects in ADO, through the use of a new C++ precompiler directive called `#import`.

The `#import` precompiler directive was first added to Visual C++ with the 5.0 release. You can use this directive to import an ActiveX DLL that has been built with the `IDispatch` interface description included in the DLL. This directive tells the Visual C++ compiler to import the DLL specified by the `#import` directive and to extract the object information from the DLL, creating a couple of header files that are automatically included in your project. These header files have the filename extensions `.TLH` and `.TLI` and are in the output directory for your project (the Debug or Release directory, the same directory where you'll find the executable application after you've compiled your project). These two files contain definitions of classes for each of the objects in the DLL that you can use in your code. The `#import` directive also tells the compiler to include the DLL as part of the project, eliminating the need to include the `.LIB` file for the DLL in your project.

You can import the ADO DLL by placing the following code at the beginning of the header file in which you are defining any database objects:

```
#define INITGUID
#import "C:\Program Files\Common Files\System\ADO\msado15.dll"
        ➥rename_namespace("ADOCG") rename("EOF", "EndOfFile")
using namespace ADOCG;
#include "icrsint.h"
```

In these four lines of directives, the first line defines a constant that needs to be defined for ADO. The second imports the ADO DLL, creating the two header files mentioned earlier. After the filename to be imported, this directive includes two attributes to the `#import` directive. The first, `rename_namespace`, renames the namespace into which the DLL has been imported. This is followed with the line following the `#import`, where the renamed namespace is specified as the one used. The second attribute, `rename`, renames an element in the header files that are created using the `#import` directive.

The reason you rename elements in these header files is to prevent conflicts with another element named elsewhere. If you examine the header file, the element specified is not renamed in the file, but when the compiler reads the file, the element is renamed. The final line includes the ADO header file, which contains the definition of some macros that you will use when writing your ADO applications.

Connecting to a Database

Before you can use any of the ADO objects, you need to initialize the COM environment for your application. You can do this by calling the `CoInitialize` API function, passing `NULL` as the only parameter, as follows:

```
::CoInitialize(NULL);
```

This enables you to make calls to ActiveX objects. If you leave out this one line of code from your application, or don't put it before you begin interacting with the objects, you get an COM error whenever you run your application.

When you are finished with all ADO activity, you also need to shut down the COM environment by calling the `CoUninitialize` function, as follows:

```
CoUninitialize();
```

This function cleans up the COM environment and prepares your application for shutting down.

Once you initialize the COM environment, you can create a connection to the database. The best way to do this is not to declare a `Connection` object variable, but to declare a `Connection` object pointer, `_ConnectionPtr`, and use it for all your interaction with the `Connection` object. Once you declare a `Connection` object pointer, you can initialize it by creating an instance of the `Connection` object, calling the `CreateInstance` function, passing it the `UUID` of the `Connection` object as its only parameter, as follows:

```
_ConnectionPtr pConn;
pConn.CreateInstance(__uuidof(Connection));
```

When you work with these objects and functions, you need to use the correct number of underscore characters in front of the various object and function names. The `_ConnectionPtr` object has only a single underscore character, whereas the `__uuidof` function has two.

Once you create the object, you can call the `Open` function to establish the connection to the database. This function takes four parameters. The first parameter is the connection definition string. This string defines the OLE DB data source for the database. It may be

an ODBC OLE DB driver, where OLE DB is sitting on top of an ODBC data source, as you'll use in your sample application. If you are using SQL Server or Oracle databases, it may be a direct connection to the OLE DB interface provided by the database itself. The second parameter is the user ID for connecting to the database. The third parameter is the password for connecting to the database. The fourth parameter is the cursor type to use with the database. These types are defined in the `msado15.tlh` header file that is created by the `#import` directive. A typical use of the `Open` function to connect to an ODBC data source that doesn't need a user ID or password is like the following:

```
pConn->Open(L"Provider=MSDASQL.1;Data Source=TYVCDB", L"", L"",
    ➥adOpenUnspecified);
```

Executing Commands and Retrieving Data

Once you have the connection open, you can use a `Command` object to pass SQL commands to the database. This is the normal method of executing SQL commands with ADO. To create a `Command` object, follow the same process that you used to create a `Connection` object. Declare a `Command` object pointer, `_CommandPtr`, and then create an instance of it using the `UUID` of the `Command` object, as follows:

```
_CommandPtr pCmd;
pCmd.CreateInstance(__uuidof(Command));
```

Once you create your `Command` object, assuming that you have already established the connection to the database, set the active connection property of the `Command` object to the open `Connection` object pointer, as follows:

```
pCmd->ActiveConnection = pConn;
```

Next, specify the SQL command to be executed by setting the `CommandText` property of the `Command` object, as follows:

```
pCmd->CommandText = "Select * from Addresses";
```

At this point, you have two options for how you execute this command and retrieve the records. The first is to call the `Command` object's `Execute` method, which will return a new `Recordset` object, which you'll want to set to a `Recordset` object pointer, as follows:

```
_RecordsetPtr pRs;
pRs = pCmd->Execute();
```

The other approach to running the command and retrieving the records is to specify that the `Command` object is the source for the records in the `Recordset`. This requires creating the `Recordset` object as follows:

```
_RecordsetPtr pRs;
pRs.CreateInstance(__uuidof(Recordset));
pRs->PutRefSource(pCmd);
```

Now, you'll need to create two `NULL variant` values to pass as the first two parameters to the `Recordset`'s `Open` method. The third parameter will be the cursor type to use, followed by the locking method to use. Finally, the fifth parameter to the `Recordset`'s `Open` method is an options flag that indicates how the database should evaluate the command being passed in. You do this with the following code:

```
// Create the variant NULL
_variant_t vNull;
vNull.vt = VT_ERROR;
vNull.scode = DISP_E_PARAMNOTFOUND;

// Open the recordset
pRs->Open(vNull, vNull, adOpenDynamic, adLockOptimistic, adCmdUnknown);
```

You could take another approach to accomplish all of the preceding tasks with only a few lines of code. Skip the use of the `Command` and `Connection` objects altogether, placing all the necessary connection information in the `Recordset`'s `Open` function. You can specify the SQL command as the first parameter and the connection information as the second parameter, instead of the two `NULL`s that you passed previously. This method reduces all of the preceding code to the following few lines:

```
_RecordsetPtr pRs;
pRs.CreateInstance(__uuidof(Recordset));
pRs->Open(_T("Provider=MSDASQL.1;Data Source=TYVCDB"),
          _T("select * from Addresses"), adOpenDynamic,
           adLockOptimistic, adCmdUnknown);
```

Although placing all of the command and connection information into the `Recordset` `Open` function is fine for a simple application, such as the one that you will build today, you are better off using the `Connection` object with any application that has more than a couple of database queries. This allows you to make a single connection to the database and use that one connection for all interaction with the database.

Navigating the `Recordset`

Once you've retrieved a set of records from the database, and you are holding them in a `Recordset` object, you'll need to navigate the set of records. This functionality is available, just as you would expect, through the `MoveFirst`, `MoveLast`, `MovePrevious`, and `MoveNext` functions. None of these functions take any parameters because they perform the functions that you would expect them to perform.

Along with these functions, the Recordset object also has two properties, BOF and EOF (which you should normally rename to prevent a collision with the default definition of EOF), which can be checked to determine if the current record in the set is beyond either end of the set of records.

Accessing Field Values

When you need to begin accessing the data values in each of the fields is where working with ADO in Visual C++ begins to get interesting. Because ADO is intended to be easy to use in Microsoft's scripting languages, VBScript and JScript, which only have variant data types, all data elements that you'll retrieve from fields in the ADO Recordset are variant values. They have to be converted into the data types that you need them to be. There are two ways of doing this. The first way is the straight-forward way of retrieving the values into a variant and then converting them, as in the following code:

```
_variant_t vFirstName;
CString strFirstName;

vFirstName = pRs->GetCollect(_variant_t("FirstName"));
vFirstName.ChangeType(VT_BSTR);
strFirstName = vFirstName.bstrVal;
```

The not-so-straight-forward way to do this is actually the better way, and in the long run, is a lot easier to work with. Microsoft has created a series of macros that perform the conversion for you and that maintain a set of variables of the records in the set. To do this, you'll define a new class to use as the interface for your record set. This class will be a descendent of the CADORecordBinding class, which is defined in the icrsint.h header file, which you included just after the #import directive. This class will not have any constructor or destructor but will have a series of macros, along with a number of variables. Each field in the set of records has two variables, an unsigned long, which is used to maintain the status of the variable, and the field variable itself. These variables must be regular C variables, and they cannot be C++ classes such as CString. A simple example of this class declaration is the following:

```
class CCustomRs :
    public CADORecordBinding
{
BEGIN_ADO_BINDING(CCustomRs)
    ADO_FIXED_LENGTH_ENTRY(1, adInteger, m_lAddressID, lAddressIDStatus,
    ➥FALSE)
    ADO_VARIABLE_LENGTH_ENTRY2(2, adVarChar, m_szFirstName,
        ➥sizeof(m_szFirstName), lFirstNameStatus, TRUE)
    ADO_FIXED_LENGTH_ENTRY(3, adDate, m_dtBirthdate, lBirthdateStatus,
    ➥TRUE)
    ADO_FIXED_LENGTH_ENTRY(4, adBoolean, m_bSendCard, lSendCardStatus,
    ➥TRUE)
```

```
END_ADO_BINDING()

public:
    LONG m_lAddressID;
    ULONG lAddressIDStatus;
    CHAR m_szFirstName[51];
    ULONG lFirstNameStatus;
    DATE m_dtBirthdate;
    ULONG lBirthdateStatus;
    VARIANT_BOOL m_bSendCard;
    ULONG lSendCardStatus;
};
```

Once you define this record layout class to match the record layout that will be returned by your database query, you can declare a variable of this class for use in your application, as follows:

```
CCustomRs m_rsRecSet;
```

Next, you need to create a pointer to an `IADORecordBinding` interface, as follows:

```
IADORecordBinding *picRs = NULL;
```

This is a pointer to a `COM` interface that is part of the ADO `Recordset` object. Once you retrieve the set of records, you need to retrieve the pointer to the `IADORecordBinding` interface and bind the custom record set class to the `Recordset` object, as in the following code:

```
if (FAILED(pRs->QueryInterface(__uuidof(IADORecordBinding), (LPVOID
➥*)&picRs)))
    _com_issue_error(E_NOINTERFACE);
picRs->BindToRecordset(&m_rsRecSet);
```

Now, as you navigate the records in the set, you just need to access the member variables of your custom record class to retrieve the current value for each field.

The `BEGIN_ADO_BINDING` and `END_ADO_BINDING` Macros

The key to the second method of accessing the data values in the record set is in the macros that are used in defining the record class. The set of macros start with the `BEGIN_ADO_BINDING` macro, which takes the class name as its only parameter. This macro sets up the structure definition that is created with the rest of the macros that follow.

The set of macros is closed by the `END_ADO_BINDING` macro. This macro doesn't take any parameters, and it wraps up the definition of the record binding structure that is created in the class. It is in the rest of the macros, which are used between these two, where the real work is done.

The ADO_FIXED_LENGTH_ENTRY Macros

The ADO_FIXED_LENGTH_ENTRY macro is used for any database fields that are fixed in size. It can be used with a date or boolean field, or even a text field that is a fixed size, with no option for any variation in the database. There are two versions of this macro; you add a 2 to the end of the name of the second version (ADO_FIXED_LENGTH_ENTRY2).

Both versions require the same first three and last parameters. The first version requires an additional parameter that is not required in the second version. The first parameter is the ordinal number of the field in the record set. This is the position in the field order as returned by the SQL query that is run to populate the record set. The second parameter is the data type of the field; the available data types are defined in the header file created by the #import directive. The third parameter is the variable into which the data value is to be copied. For the first version of the macro, the fourth parameter is the variable for the field status (the unsigned long that you defined with the variable for the actual value). The last variable is a boolean that specifies whether this field can be modified.

The ADO_NUMERIC_ENTRY Macros

You use the ADO_NUMERIC_ENTRY macros with numeric fields only. They are similar to the ADO_FIXED_LENGTH_ENTRY macros in that there are two different versions of the macro, named in the same way. In these macros, the first five parameters are the same in both versions, along with the final parameter. Like with the ADO_FIXED_LENGTH_ENTRY macros, the first version has an additional parameter that is not used in the second version.

The first three parameters for the ADO_NUMERIC_ENTRY macros are the same as those for the ADO_FIXED_LENGTH_ENTRY macros, as are the last parameter and the next to last parameter for the first version. It is the fourth and fifth parameters that are unique to these macros. The fourth parameter specifies the precision of the value in this field of the record set. The fifth parameter specifies the scale of the value. Both of these parameters are crucial in correctly converting the value to and from a variant data type.

The ADO_VARIABLE_LENGTH_ENTRY Macros

The final series of macros is the ADO_VARIABLE_LENGTH_ENTRY macros. You use this series of macros with database fields that are likely to vary in length. With a SQL-based database, you want to use this series of macros with any varchar (variable-length character string) columns. There are three versions of this macro. In all three versions, the first four parameters are the same, and the final parameter is the same. It is the parameters between them that vary.

The first parameter is the ordinal position of the column in the record set as returned by the SQL query. The second parameter is the data type. The third parameter is the variable in which the data value should be placed. The fourth parameter for all versions of the macro is the size of the variable into which the value is to be placed. This prevents the data from being written past the end of the variable that you defined for it to be placed in. As with the previous macros, the final parameter specifies whether the field is updateable.

In the first version of this macro, there are two parameters between the fourth and final parameters. The second version of this macro only has the first of these two parameters, and the third version only has the second of these two parameters. The first of these two parameters is the status variable for use with this field. The second of these two parameters is the length of the field in the database. The preceding example used the second version of this macro.

Updating Records

When you need to update values in a record in the recordset, how you handle it depends on which of the two methods you used to retrieve the data elements from the recordset. If you retrieved each field and converted it from a `variant` yourself, you need to update each individual field that has been changed. The update is done using the `Recordset` object's `Update` method, which takes two variables, the field being updated and the new value for the field. You could make this update using the following code:

```
_variant_t vName, vValue;
vName.SetString("FirstName");
vValue.SetString("John");
pRs->Update(vName, vValue);
```

If you created your record class and bound it to the recordset, updating the record is a little simpler. Once you have copied the new values into the variables in the record class, you can call the record-bound version of the `Update` function, as in the following:

```
picRs->Update(&m_rsRecSet);
```

This updates the record in the `Recordset` object to be updated with the values in the record class that you have bound to the set.

Adding and Deleting

Adding and deleting records from an ADO recordset is similar to how you accomplish it in other database access technologies. However, there are some slight subtleties to how you perform the addition of new records.

For deleting the current record, you can call the `Recordset` object's `Delete` method. This method requires a single parameter that specifies how the delete is supposed to be done. Most likely, you'll pass the `adAffectCurrent` value so that only the current record in the recordset is deleted, as in the following code:

```
pRs->Delete(adAffectCurrent);
pRs->MovePrevious();
```

As with any other database access technology, once you've deleted the current record, there is no current record, so you need to navigate to another record before allowing the user to do anything else.

When you are adding a new record, you can call the `Recordset` object's `AddNew` method. Once you have added a new record, the new record is the current record in the record set. If you check the variables in the record class that you created, you'll find that they are all empty. However, you cannot just begin entering data values into these fields. To allow the user to immediately enter the various data elements in the new record, you'll blank out the values in the record class and pass this variable as the only parameter to the Add New class. You need to call it through the record-binding interface pointer, as in the following example:

```
CString strBlank = " ";
COleDateTime dtBlank;

m_rsRecSet.m_lAddressID = 0;
strcpy(m_rsRecSet.m_szFirstName, (LPCTSTR)strBlank);
m_rsRecSet.m_dtBirthdate = (DATE)dtBlank;
m_rsRecSet.m_bSendCard = VARIANT_FALSE;
picRs->AddNew(&m_rsRecSet);
```

This allows you to provide the user with a blank record, ready for editing. Once the user has entered all the various values in the record, copy all these values back to the record variable. Then, call the `Update` method to save the record.

Closing the `Recordset` and `Connection` Objects

Once you finish working with a record set, you'll close the record set by calling the `Close` method, as follows:

```
pRs->Close();
```

Once you finish all database interaction for the entire application, you'll also close the connection to the database by calling the `Connection` object's `Close` method:

```
pConn->Close();
```

Building a Database Application Using ADO

The sample application that you will build today is another simple database application, basically the same as the one you built yesterday. You'll use ADO to retrieve a set of records from an Access database, providing functionality to navigate the record set. The user will be able to make changes to the data in the record set, and those changes will be reflected in the database as well. The user will also be able to add new records to the record set and delete records as desired. You will accomplish all of this using ADO as the means of accessing the database, which will go through the ODBC driver that was configured yesterday.

Creating the Application Shell

The application that you will build today will be an SDI-style application. As with several other sample applications that you build in the course of reading this book, everything that you do in today's application is just as applicable to an MDI or dialog-style application. To start the application, you'll use the MFC AppWizard to build the application shell, using most of the SDI-style application default settings.

To start your application, create a new AppWizard project, naming the project something appropriate, such as `DbAdo`. Specify on the first panel of the AppWizard that you are building an SDI-style application. Accept all the default settings for steps 2 through 5, being sure to leave the second step stating that you want no database support included in the application. On the final AppWizard step, specify that the view class should be inherited from the `CFormView` class.

Once you finish creating your application shell, design the main dialog form for use in your application. Add the standard controls for each of the fields in the Addresses table from the database you used yesterday (or if you used a different database yesterday, add controls for all the fields in the table that you used), as shown in Figure 15.6. Configure the controls using the properties listed in Table 15.1.

If you want to save a little time when building the example, you can leave out most of the controls and database fields from the application. The key fields that you'll need to include on the screen are ID, First and Last Names, Birthdate, and Send Card. If you want to leave out the other fields from the application, that's fine. You will need to include these fields in the `CCustomRs` class that you create in this chapter.

FIGURE 15.6.
The main form layout.

TABLE 15.1. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`Address ID`
Edit Box	ID	`IDC_EDIT_ADDRESSID`
Static Text	ID	`IDC_STATIC`
	Caption	`First Name`
Edit Box	ID	`IDC_EDIT_FIRSTNAME`
Static Text	ID	`IDC_STATIC`
	Caption	`Last Name`
Edit Box	ID	`IDC_EDIT_LASTNAME`
Static Text	ID	`IDC_STATIC`
	Caption	`Spouse Name`
Edit Box	ID	`IDC_EDIT_SPOUSENAME`
Static Text	ID	`IDC_STATIC`
	Caption	`Address`
Edit Box	ID	`IDC_EDIT_ADDRESS`

continues

TABLE 15.1. CONTINUED

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`City`
Edit Box	ID	`IDC_EDIT_CITY`
Static Text	ID	`IDC_STATIC`
	Caption	`State Or Province`
Edit Box	ID	`IDC_EDIT_STATEORPROVINCE`
Static Text	ID	`IDC_STATIC`
	Caption	`Postal Code`
Edit Box	ID	`IDC_EDIT_POSTALCODE`
Static Text	ID	`IDC_STATIC`
	Caption	`Country`
Edit Box	ID	`IDC_EDIT_COUNTRY`
Static Text	ID	`IDC_STATIC`
	Caption	`Email Address`
Edit Box	ID	`IDC_EDIT_EMAILADDRESS`
Static Text	ID	`IDC_STATIC`
	Caption	`Home Phone`
Edit Box	ID	`IDC_EDIT_HOMEPHONE`
Static Text	ID	`IDC_STATIC`
	Caption	`Work Phone`
Edit Box	ID	`IDC_EDIT_WORKPHONE`
Static Text	ID	`IDC_STATIC`
	Caption	`Work Extension`
Edit Box	ID	`IDC_EDIT_WORKEXTENSION`
Static Text	ID	`IDC_STATIC`
	Caption	`Fax Number`
Edit Box	ID	`IDC_EDIT_FAXNUMBER`
Static Text	ID	`IDC_STATIC`
	Caption	`Birthdate`
Edit Box	ID	`IDC_EDIT_BIRTHDATE`
Static Text	ID	`IDC_STATIC`
	Caption	`Send Card`

Object	*Property*	*Setting*
Check Box	ID	`IDC_CHECK_SENDCARD`
Static Text	ID	`IDC_STATIC`
	Caption	`Notes`
Edit Box	ID	`IDC_EDIT_NOTES`

Once you add all of the controls to the form, use the Class Wizard to attach variables to each of these controls, as specified in Table 15.2. The variables should match the data types of the columns in the database that the control will be used to display.

TABLE 15.2. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
`IDC_CHECK_SENDCARD`	`m_bSendCard`	`Value`	`BOOL`
`IDC_EDIT_ADDRESS`	`m_strAddress`	`Value`	`CString`
`IDC_EDIT_ADDRESSID`	`m_lAddressID`	`Value`	`long`
`IDC_EDIT_BIRTHDATE`	`m_oledtBirthdate`	`Value`	`COleDateTime`
`IDC_EDIT_CITY`	`m_strCity`	`Value`	`CString`
`IDC_EDIT_COUNTRY`	`m_strCountry`	`Value`	`CString`
`IDC_EDIT_EMAILADDRESS`	`m_strEmailAddress`	`Value`	`CString`
`IDC_EDIT_FAXNUMBER`	`m_strFaxNumber`	`Value`	`CString`
`IDC_EDIT_FIRSTNAME`	`m_strFirstName`	`Value`	`CString`
`IDC_EDIT_HOMEPHONE`	`m_strHomePhone`	`Value`	`CString`
`IDC_EDIT_LASTNAME`	`m_strLastName`	`Value`	`CString`
`IDC_EDIT_NOTES`	`m_strNotes`	`Value`	`CString`
`IDC_EDIT_POSTALCODE`	`m_strPostalCode`	`Value`	`CString`
`IDC_EDIT_SPOUSENAME`	`m_strSpouseName`	`Value`	`CString`
`IDC_EDIT_STATEORPROVINCE`	`m_strStateOrProvince`	`Value`	`CString`
`IDC_EDIT_WORKEXTENSION`	`m_strWorkExtension`	`Value`	`CString`
`IDC_EDIT_WORKPHONE`	`m_strWorkPhone`	`Value`	`CString`

Building a Custom Record Class

Before you go any further in building your application, you need to create your custom record class that you will bind to the record set. This class will need public variables for each of the columns in the database table that you are selecting, as well as status

variables for each of these columns. You'll also build the set of macros to exchange the column values between the record set and the class variables. To create this class, create a new class using the same method you used in previous days, specifying that a generic class. Specify a suitable class name, such as `CCustomRs`, and specify the base class as `CADORecordBinding` with public access.

Once you have created your new class, delete the constructor and destructor functions from both the header and source code files for the new class. Edit the header file for your new class, importing the ADO DLL and filling in the macros and variables, as in Listing 15.1.

LISTING 15.1. THE CUSTOM RECORD CLASS.

```
 1: #define INITGUID
 2: #import "C:\Program Files\Common Files\System\ADO\msado15.dll"
            ➥rename_namespace("ADOCG") rename("EOF", "EndOfFile")
 3: using namespace ADOCG;
 4: #include "icrsint.h"
 5:
 6: class CCustomRs :
 7:     public CADORecordBinding
 8: {
 9: BEGIN_ADO_BINDING(CCustomRs)
10:     ADO_FIXED_LENGTH_ENTRY(1, adInteger, m_lAddressID,
        ➥lAddressIDStatus,FALSE)
11:     ADO_VARIABLE_LENGTH_ENTRY2(2, adVarChar, m_szFirstName,
            ➥sizeof(m_szFirstName), lFirstNameStatus, TRUE)
12:     ADO_VARIABLE_LENGTH_ENTRY2(3, adVarChar, m_szLastName,
            ➥sizeof(m_szLastName), lLastNameStatus, TRUE)
13:     ADO_VARIABLE_LENGTH_ENTRY2(4, adVarChar, m_szSpouseName,
            ➥sizeof(m_szSpouseName), lSpouseNameStatus, TRUE)
14:     ADO_VARIABLE_LENGTH_ENTRY2(5, adVarChar, m_szAddress,
            ➥sizeof(m_szAddress), lAddressStatus, TRUE)
15:     ADO_VARIABLE_LENGTH_ENTRY2(6, adVarChar, m_szCity,
        ➥sizeof(m_szCity),lCityStatus, TRUE)
16:     ADO_VARIABLE_LENGTH_ENTRY2(7, adVarChar, m_szStateOrProvince,
            ➥sizeof(m_szStateOrProvince), lStateOrProvinceStatus, TRUE)
17:     ADO_VARIABLE_LENGTH_ENTRY2(8, adVarChar, m_szPostalCode,
            ➥sizeof(m_szPostalCode), lPostalCodeStatus, TRUE)
18:     ADO_VARIABLE_LENGTH_ENTRY2(9, adVarChar, m_szCountry,
            ➥sizeof(m_szCountry), lCountryStatus, TRUE)
19:     ADO_VARIABLE_LENGTH_ENTRY2(10, adVarChar, m_szEmailAddress,
            ➥sizeof(m_szEmailAddress), lEmailAddressStatus, TRUE)
20:     ADO_VARIABLE_LENGTH_ENTRY2(11, adVarChar, m_szHomePhone,
            ➥sizeof(m_szHomePhone), lHomePhoneStatus, TRUE)
21:     ADO_VARIABLE_LENGTH_ENTRY2(12, adVarChar, m_szWorkPhone,
            ➥sizeof(m_szWorkPhone), lWorkPhoneStatus, TRUE)
```

```
22:     ADO_VARIABLE_LENGTH_ENTRY2(13, adVarChar, m_szWorkExtension,
            ➥sizeof(m_szWorkExtension), lWorkExtensionStatus, TRUE)
23:     ADO_VARIABLE_LENGTH_ENTRY2(14, adVarChar, m_szFaxNumber,
            ➥sizeof(m_szFaxNumber), lFaxNumberStatus, TRUE)
24:     ADO_FIXED_LENGTH_ENTRY(15, adDate, m_dtBirthdate,
        ➥lBirthdateStatus,TRUE)
25:     ADO_FIXED_LENGTH_ENTRY(16, adBoolean, m_bSendCard,
        ➥lSendCardStatus,TRUE)
26:     ADO_VARIABLE_LENGTH_ENTRY2(17, adLongVarChar, m_szNotes,
            ➥sizeof(m_szNotes), lNotesStatus, TRUE)
27: END_ADO_BINDING()
28:
29: public:
30:     LONG m_lAddressID;
31:     ULONG lAddressIDStatus;
32:     CHAR m_szFirstName[51];
33:     ULONG lFirstNameStatus;
34:     CHAR m_szLastName[51];
35:     ULONG lLastNameStatus;
36:     CHAR m_szSpouseName[51];
37:     ULONG lSpouseNameStatus;
38:     CHAR m_szAddress[256];
39:     ULONG lAddressStatus;
40:     CHAR m_szCity[51];
41:     ULONG lCityStatus;
42:     CHAR m_szStateOrProvince[21];
43:     ULONG lStateOrProvinceStatus;
44:     CHAR m_szPostalCode[21];
45:     ULONG lPostalCodeStatus;
46:     CHAR m_szCountry[51];
47:     ULONG lCountryStatus;
48:     CHAR m_szEmailAddress[51];
49:     ULONG lEmailAddressStatus;
50:     CHAR m_szHomePhone[31];
51:     ULONG lHomePhoneStatus;
52:     CHAR m_szWorkPhone[31];
53:     ULONG lWorkPhoneStatus;
54:     CHAR m_szWorkExtension[21];
55:     ULONG lWorkExtensionStatus;
56:     CHAR m_szFaxNumber[31];
57:     ULONG lFaxNumberStatus;
58:     DATE m_dtBirthdate;
59:     ULONG lBirthdateStatus;
60:     VARIANT_BOOL m_bSendCard;
61:     ULONG lSendCardStatus;
62:     CHAR m_szNotes[65536];
63:     ULONG lNotesStatus;
64: };
```

Once you've created this class, you need to add a variable to the document class. Add a new member variable to the document class, specifying the variable type as `CCustomRs`, the name as `m_rsRecSet`, and the access as private. You'll also need to include the custom record class header file in the document source code file, as in Listing 15.2.

LISTING 15.2. THE DOCUMENT SOURCE CODE INCLUDES.

```
1: // dbadoDoc.cpp : implementation of the CDbAdoDoc class
2: //
3:
4: #include "stdafx.h"
5: #include "dbado.h"
6:
7: #include "CustomRs.h"
8: #include "dbadoDoc.h"
9: #include "dbadoView.h"
```

Another detail that you need to attend to before going any further is providing a way for the view to get a pointer to the record class from the document class. This function should return a pointer to the record class variable. To add this function to your application, add a new member function to the document class, specifying the function type as `CCustomRs*`, the function declaration as `GetRecSet`, and the function access as public. Edit this function, adding the code in Listing 15.3.

LISTING 15.3. THE `CDbAdoDoc` `GetRecSet` FUNCTION.

```
1: CCustomRs* CDbAdoDoc::GetRecSet()
2: {
3:     // Return a pointer to the record object
4:     return &m_rsRecSet;
5: }
```

One last piece of functionality that you'll add before getting to the real heart of ADO programming is the function for reporting ADO and database errors. This function will display a message to the user, reporting that an error occurred and displaying the error code and error message for the user. To add this function to your application, add a new member function to your document class. Specify the function type as `void`, the function declaration as `GenerateError(HRESULT hr, PWSTR pwszDescription)`, and the access as public. Edit the function, entering the code in Listing 15.4.

LISTING 15.4. THE `CDbAdoDoc GenerateError` FUNCTION.

```
 1: void CDbAdoDoc::GenerateError(HRESULT hr, PWSTR pwszDescription)
 2: {
 3:     CString strError;
 4:
 5:     // Format and display the error message
 6:     strError.Format("Run-time error '%d (%x)'", hr, hr);
 7:     strError += "\n\n";
 8:     strError += pwszDescription;
 9:
10:     AfxMessageBox(strError);
11: }
```

Connecting and Retrieving Data

You can perform all of the connecting to the database and retrieving the record set in the `OnNewDocument` function in the document class. Before you can add this functionality, you need to add a few more variables to the document class. You'll need a `Recordset` object pointer, an `IADORecordBinding` interface pointer, a couple of string variables for holding the database connection string, and the SQL command to execute to populate the record set. Add all of these variables to the document class as specified in Table 15.3.

TABLE 15.3. DOCUMENT CLASS MEMBER VARIABLES.

Name	*Type*	*Access*
`m_pRs`	`_RecordsetPtr`	Private
`m_piAdoRecordBinding`	`IADORecordBinding*`	Private
`m_strConnection`	`CString`	Private
`m_strCmdText`	`CString`	Private

In the `OnNewDocument` function, you'll perform a series of steps for connecting and retrieving the record set. First, you'll set the strings for the database connection and the SQL command to be run. Next, you'll initialize the `COM` environment and initialize the two pointers so that they are both `NULL`. You'll create the `Recordset` object using the `CreateInstance` function. Open the `Recordset`, connecting to the database and running the SQL command at the same time. Bind the record class to the record set using the `IADORecordBinding` interface pointer. Finally, tell the view class to refresh the bound data, displaying the initial record for the user using a view class function that you'll add in a little while. To add all this functionality, edit the `OnNewDocument` function in the document class, adding the code starting with line 8 in Listing 15.5.

LISTING 15.5. THE `CDbAdoDoc` `OnNewDocument` FUNCTION.

```
 1: BOOL CDbAdoDoc::OnNewDocument()
 2: {
 3:     if (!CDocument::OnNewDocument())
 4:         return FALSE;
 5:
 6:     // TODO: add reinitialization code here
 7:     // (SDI documents will reuse this document)
 8:     // Set the connection and SQL command strings
 9:     m_strConnection = _T("Provider=MSDASQL.1;Data Source=TYVCDB");
10:     m_strCmdText = _T("select * from Addresses");
11:
12:     // Initialize the Recordset and binding pointers
13:     m_pRs = NULL;
14:     m_piAdoRecordBinding = NULL;
15:     // Initialize the COM environment
16:     ::CoInitialize(NULL);
17:     try
18:     {
19:         // Create the record set object
20:         m_pRs.CreateInstance(__uuidof(Recordset));
21:
22:         // Open the record set object
23:         m_pRs->Open((LPCTSTR)m_strCmdText, (LPCTSTR)m_strConnection,
24:             adOpenDynamic, adLockOptimistic, adCmdUnknown);
25:
26:         // Get a pointer to the record binding interface
27:         if (FAILED(m_pRs->QueryInterface(__uuidof(IADORecordBinding),
28:                 (LPVOID *)&m_piAdoRecordBinding)))
29:             _com_issue_error(E_NOINTERFACE);
30:         // Bind the record class to the record set
31:         m_piAdoRecordBinding->BindToRecordset(&m_rsRecSet);
32:
33:         // Get a pointer to the view
34:         POSITION pos = GetFirstViewPosition();
35:         CDbAdoView* pView = (CDbAdoView*)GetNextView(pos);
36:         if (pView)
37:             // Sync the data set with the form
38:             pView->RefreshBoundData();
39:     }
40:     // Any errors?
41:     catch (_com_error &e)
42:     {
43:         // Display the error
44:         GenerateError(e.Error(), e.Description());
45:     }
46:
47:     return TRUE;
48: }
```

Before moving any further, it's a good idea to make sure that you add all the code necessary to clean up as your application is closing. You need to close the record set and release the pointer to the record binding interface. You'll also shut down the COM environment. To add all this functionality to your application, add a function to the `DeleteContents` event message in the document class. Edit this function, adding the code in Listing 15.6.

LISTING 15.6. THE `CDbAdoDoc` `DeleteContents` FUNCTION.

```
 1: void CDbAdoDoc::DeleteContents()
 2: {
 3:     // TODO: Add your specialized code here and/or call the base class
 4:     // Close the record set
 5:     if (m_pRs)
 6:         m_pRs->Close();
 7:     // Do we have a valid pointer to the record binding?
 8:     if (m_piAdoRecordBinding)
 9:         // Release it
10:         m_piAdoRecordBinding->Release();
11:     // Set the record set pointer to NULL
12:     m_pRs = NULL;
13:
14:     // Shut down the COM environment
15:     CoUninitialize();
16:
17:     CDocument::DeleteContents();
18: }
```

Populating the Form

To display the record column values for the user, you'll add a function for copying the values from the record class to the view variables. This function first needs to get a pointer to the record class from the document class. Next, it will check the status of each individual field in the record class to make sure that it's okay to copy, and then it will copy the value. Once all values have been copied, you can call `UpdateData` to display the values in the controls on the form. To add this functionality to your application, add a new member function to the view class. Specify the function type as `void`, the function declaration as `RefreshBoundData`, and the access as public. Edit this new function, adding the code in Listing 15.7.

LISTING 15.7. THE CDbAdoView RefreshBoundData FUNCTION.

```
 1: void CDbAdoView::RefreshBoundData()
 2: {
 3:     CCustomRs* pRs;
 4:
 5:     // Get a pointer to the document object
 6:     pRs = GetDocument()->GetRecSet();
 7:
 8:     // Is the field OK
 9:     if (adFldOK == pRs->lAddressIDStatus)
10:         // Copy the value
11:         m_lAddressID = pRs->m_lAddressID;
12:     else
13:         // Otherwise, set the value to 0
14:         m_lAddressID = 0;
15:     // Is the field OK
16:     if (adFldOK == pRs->lFirstNameStatus)
17:         // Copy the value
18:         m_strFirstName = pRs->m_szFirstName;
19:     else
20:         // Otherwise, set the value to 0
21:         m_strFirstName = _T("");
22:     if (adFldOK == pRs->lLastNameStatus)
23:         m_strLastName = pRs->m_szLastName;
24:     else
25:         m_strLastName = _T("");
26:     if (adFldOK == pRs->lSpouseNameStatus)
27:         m_strSpouseName = pRs->m_szSpouseName;
28:     else
29:         m_strSpouseName = _T("");
30:     if (adFldOK == pRs->lAddressStatus)
31:         m_strAddress = pRs->m_szAddress;
32:     else
33:         m_strAddress = _T("");
34:     if (adFldOK == pRs->lCityStatus)
35:         m_strCity = pRs->m_szCity;
36:     else
37:         m_strCity = _T("");
38:     if (adFldOK == pRs->lStateOrProvinceStatus)
39:         m_strStateOrProvince = pRs->m_szStateOrProvince;
40:     else
41:         m_strStateOrProvince = _T("");
42:     if (adFldOK == pRs->lPostalCodeStatus)
43:         m_strPostalCode = pRs->m_szPostalCode;
44:     else
45:         m_strPostalCode = _T("");
46:     if (adFldOK == pRs->lCountryStatus)
47:         m_strCountry = pRs->m_szCountry;
48:     else
```

```
49:             m_strCountry = _T("");
50:         if (adFldOK == pRs->lEmailAddressStatus)
51:             m_strEmailAddress = pRs->m_szEmailAddress;
52:         else
53:             m_strEmailAddress = _T("");
54:         if (adFldOK == pRs->lHomePhoneStatus)
55:             m_strHomePhone = pRs->m_szHomePhone;
56:         else
57:             m_strHomePhone = _T("");
58:         if (adFldOK == pRs->lWorkPhoneStatus)
59:             m_strWorkPhone = pRs->m_szWorkPhone;
60:         else
61:             m_strWorkPhone = _T("");
62:         if (adFldOK == pRs->lWorkExtensionStatus)
63:             m_strWorkExtension = pRs->m_szWorkExtension;
64:         else
65:             m_strWorkExtension = _T("");
66:         if (adFldOK == pRs->lFaxNumberStatus)
67:             m_strFaxNumber = pRs->m_szFaxNumber;
68:         else
69:             m_strFaxNumber = _T("");
70:         if (adFldOK == pRs->lBirthdateStatus)
71:             m_oledtBirthdate = pRs->m_dtBirthdate;
72:         else
73:             m_oledtBirthdate = 0L;
74:         if (adFldOK == pRs->lSendCardStatus)
75:             m_bSendCard = VARIANT_FALSE == pRs->m_bSendCard ? FALSE :
                ➥TRUE;
76:         else
77:             m_bSendCard = FALSE;
78:         if (adFldOK == pRs->lNotesStatus)
79:             m_strNotes = pRs->m_szNotes;
80:         else
81:             m_strNotes = _T("");
82:
83:         // Sync the data with the controls
84:         UpdateData(FALSE);
85: }
```

Because you are working directly with the custom record class that you created in this function, you must include the header file for your custom record class in the view class source file, just as you did with the document class source file.

Saving Updates

When you need to copy changes back to the record set, reverse the process of copying data from the controls on the form to the variables in the record class. You could take the approach of copying all values, regardless of whether their values have changed, or you could compare the two values to determine which have changed and need to be copied back. Call the function that does this before navigating to any other records in the record set so that any changes that the user has made are saved to the database. To add this functionality to your application, add a new member function to the view class. Specify the function type as `void`, the function declaration as `UpdateBoundData`, and the access as private. Edit the function, adding the code in Listing 15.8.

LISTING 15.8. THE `CDbAdoView` `UpdateBoundData` FUNCTION.

```
 1: void CDbAdoView::UpdateBoundData()
 2: {
 3:     CCustomRs* pRs;
 4:
 5:     // Get a pointer to the document
 6:     pRs = GetDocument()->GetRecSet();
 7:
 8:     // Sync the controls with the variables
 9:     UpdateData(TRUE);
10:
11:     // Has the field changed? If so, copy the value back
12:     if (m_lAddressID != pRs->m_lAddressID)
13:         pRs->m_lAddressID = m_lAddressID;
14:     if (m_strFirstName != pRs->m_szFirstName)
15:         strcpy(pRs->m_szFirstName, (LPCTSTR)m_strFirstName);
16:     if (m_strLastName != pRs->m_szLastName)
17:         strcpy(pRs->m_szLastName, (LPCTSTR)m_strLastName);
18:     if (m_strSpouseName != pRs->m_szSpouseName)
19:         strcpy(pRs->m_szSpouseName, (LPCTSTR)m_strSpouseName);
20:     if (m_strAddress != pRs->m_szAddress)
21:         strcpy(pRs->m_szAddress, (LPCTSTR)m_strAddress);
22:     if (m_strCity != pRs->m_szCity)
23:         strcpy(pRs->m_szCity, (LPCTSTR)m_strCity);
24:     if (m_strStateOrProvince != pRs->m_szStateOrProvince)
25:         strcpy(pRs->m_szStateOrProvince,
            ➥(LPCTSTR)m_strStateOrProvince);
26:     if (m_strPostalCode != pRs->m_szPostalCode)
27:         strcpy(pRs->m_szPostalCode, (LPCTSTR)m_strPostalCode);
28:     if (m_strCountry != pRs->m_szCountry)
29:         strcpy(pRs->m_szCountry, (LPCTSTR)m_strCountry);
30:     if (m_strEmailAddress != pRs->m_szEmailAddress)
31:         strcpy(pRs->m_szEmailAddress, (LPCTSTR)m_strEmailAddress);
32:     if (m_strHomePhone != pRs->m_szHomePhone)
```

```
33:         strcpy(pRs->m_szHomePhone, (LPCTSTR)m_strHomePhone);
34:     if (m_strWorkPhone != pRs->m_szWorkPhone)
35:         strcpy(pRs->m_szWorkPhone, (LPCTSTR)m_strWorkPhone);
36:     if (m_strWorkExtension != pRs->m_szWorkExtension)
37:         strcpy(pRs->m_szWorkExtension, (LPCTSTR)m_strWorkExtension);
38:     if (m_strFaxNumber != pRs->m_szFaxNumber)
39:         strcpy(pRs->m_szFaxNumber, (LPCTSTR)m_strFaxNumber);
40:     if (((DATE)m_oledtBirthdate) != pRs->m_dtBirthdate)
41:         pRs->m_dtBirthdate = (DATE)m_oledtBirthdate;
42:     if (m_bSendCard == TRUE)
43:         pRs->m_bSendCard = VARIANT_TRUE;
44:     else
45:         pRs->m_bSendCard = VARIANT_FALSE;
46:     if (m_strNotes != pRs->m_szNotes)
47:         strcpy(pRs->m_szNotes, (LPCTSTR)m_strNotes);
48: }
```

Navigating the Record Set

For navigating the record set, add a series of menus for each of the four basic navigation choices: first, previous, next, and last. Because the `Recordset` object and the record-binding interface pointers are in the document object, the event messages for these menus must be passed to the document class to update the current record and then to navigate to the selected record. However, the view class needs to receive the event message first because it needs to copy back any changed values from the controls on the form before the update is performed. Once the navigation is complete, the view also needs to update the form with the new record's column values. Looking at the sequence of where the event message needs to be passed, it makes the most sense to add the event message handler to the view class, and from there, call the event message handler for the document class.

To add this functionality to your application, add the four menu entries and the corresponding toolbar buttons. Using the Class Wizard, add a event message handler function to the view class for the command event for all four of these menus. Edit the event function for the Move First menu, adding the code in Listing 15.9.

LISTING 15.9. THE `CDbAdoView` `OnDataFirst` FUNCTION.

```
 1: void CDbAdoView::OnDataFirst()
 2: {
 3:     // TODO: Add your command handler code here
 4:     // Update the current record
 5:     UpdateBoundData();
```

continues

LISTING 15.9. CONTINUED

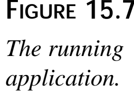
```
 6:     // Navigate to the first record
 7:     GetDocument()->MoveFirst();
 8:     // Refresh the form with the new record's data
 9:     RefreshBoundData();
10: }
```

Now add the `MoveFirst` function to the document class and perform all the actual record set functionality for this function. To add this, add a member function to the document class in your application. Specify the function type as `void`, the declaration as `MoveFirst`, and the access as public. Edit this function, adding the code in Listing 15.10.

LISTING 15.10. THE CDBADODOC MOVEFIRST FUNCTION.

```
 1: void CDbAdoDoc::MoveFirst()
 2: {
 3:     try
 4:     {
 5:         // Update the current record
 6:         m_piAdoRecordBinding->Update(&m_rsRecSet);
 7:         // Move to the first record
 8:         m_pRs->MoveFirst();
 9:     }
10:     // Any errors?
11:     catch (_com_error &e)
12:     {
13:         // Generate the error message
14:         GenerateError(e.Error(), e.Description());
15:     }
16: }
```

Edit and add the same set of functions to the view and document classes for the `MovePrevious`, `MoveNext`, and `MoveLast` ADO functions. Once you've added all these functions, you should be ready to compile and run your application. Your application will be capable of opening the Addresses database table and presenting you with each individual record, which you can edit and update, as in Figure 15.7.



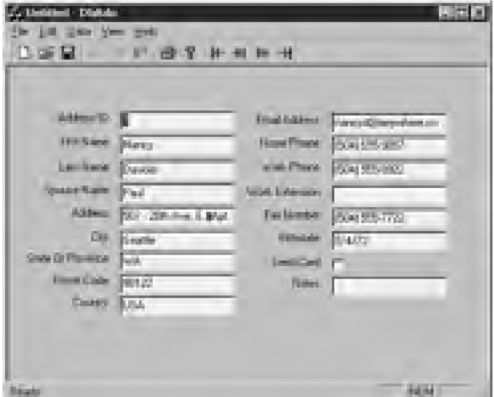


FIGURE 15.7.
The running application.

Adding New Records

Now that you are able to retrieve and navigate the set of records in the database table, it would be nice if you could add some new records to the table. You can add this functionality in exactly the same fashion that you added the navigation functionality. Add a menu, trigger an event function in the view class from the menu, update the current record values back to the record set, call a function in the document class, and refresh the current record from the record set. As far as the menu and the view class are concerned, the only difference between this functionality and any of the navigation menus and functions is the ID of the menu and the name of the functions that are called, just as with the different navigation functions. It's in the document function where things begin to diverge just a little.

In the document class function for adding a new record, once you've updated the current record, you'll make sure that adding a new record is an option. If it is, then you'll build an empty record and add it to the record set. Once you've added the empty record, navigate to the last record in the set because this will be the new record. At this point, you can exit this function and let the view class refresh the form with the data values from the new, empty record.

To add this functionality to your application, add a new menu entry for adding a new record. Add a command event-handler function to the view class for this new menu, adding the same code to the function as you did with the navigation functions, but call the `AddNew` function in the document class. Now, add the `AddNew` function to the document class. Add a new member function to the document class, specifying the type as `void`, the declaration as `AddNew`, and the access as public. Edit the function, adding the code in Listing 15.11.

LISTING 15.11. THE `CDbAdoDoc AddNew` FUNCTION.

```
 1: void CDbAdoDoc::AddNew()
 2: {
 3:     try
 4:     {
 5:         // Update the current record
 6:         m_piAdoRecordBinding->Update(&m_rsRecSet);
 7:         // Can we add a new record?
 8:         if (m_pRs->Supports(adAddNew))
 9:         {
10:             // Create a blank record
11:             CreateBlankRecord();
12:             // Add the blank record
13:             m_piAdoRecordBinding->AddNew(&m_rsRecSet);
14:             // Move to the last record
15:             m_pRs->MoveLast();
16:         }
17:     }
18:     // Any errors?
19:     catch (_com_error &e)
20:     {
21:         // Generate an error message
22:         GenerateError(e.Error(), e.Description());
23:     }
24: }
```

Now add the function that creates the blank record. In this function, you'll set each of the field variables in the record class to an almost empty string. To add this function to your class, add a new member function to the document class. Specify its type as `void`, its declaration as `CreateBlankRecord`, and its access as private. Edit this new function, adding the code in Listing 15.12.

LISTING 15.12. THE `CDbAdoDoc CreateBlankRecord` FUNCTION.

```
 1: void CDbAdoDoc::CreateBlankRecord()
 2: {
 3:     // Create the blank values to be used
 4:     CString strBlank = " ";
 5:     COleDateTime dtBlank;
 6:
 7:     // Set each of the values in the record object
 8:     m_rsRecSet.m_lAddressID = 0;
 9:     strcpy(m_rsRecSet.m_szFirstName, (LPCTSTR)strBlank);
10:     strcpy(m_rsRecSet.m_szLastName, (LPCTSTR)strBlank);
```

```
11:     strcpy(m_rsRecSet.m_szSpouseName, (LPCTSTR)strBlank);
12:     strcpy(m_rsRecSet.m_szAddress, (LPCTSTR)strBlank);
13:     strcpy(m_rsRecSet.m_szCity, (LPCTSTR)strBlank);
14:     strcpy(m_rsRecSet.m_szStateOrProvince, (LPCTSTR)strBlank);
15:     strcpy(m_rsRecSet.m_szPostalCode, (LPCTSTR)strBlank);
16:     strcpy(m_rsRecSet.m_szCountry, (LPCTSTR)strBlank);
17:     strcpy(m_rsRecSet.m_szEmailAddress, (LPCTSTR)strBlank);
18:     strcpy(m_rsRecSet.m_szHomePhone, (LPCTSTR)strBlank);
19:     strcpy(m_rsRecSet.m_szWorkPhone, (LPCTSTR)strBlank);
20:     strcpy(m_rsRecSet.m_szWorkExtension, (LPCTSTR)strBlank);
21:     strcpy(m_rsRecSet.m_szFaxNumber, (LPCTSTR)strBlank);
22:     m_rsRecSet.m_dtBirthdate = (DATE)dtBlank;
23:     m_rsRecSet.m_bSendCard = VARIANT_FALSE;
24:     strcpy(m_rsRecSet.m_szNotes, (LPCTSTR)strBlank);
25: }
```

If you compile and run your application, you should be able to insert and edit new records in the database table.

Deleting Records

The final piece of functionality that you'll add to your application is the ability to delete the current record from the set. This function can follow the same form as all the navigation and add functions with a menu entry calling an event-handler function in the view class. The function in the view class can even follow the same set of code that you used in these previous functions, updating the current record, calling the corresponding function in the document class, and then refreshing the current record to the form.

In the document class function, the record deletion should follow almost the same path that you took for adding a new record. Update the current record, check to see if it's possible to delete the current record, check with the user to verify that he wants to delete the current record, and then call the `Delete` function and navigate to another record in the set.

To add this functionality to your application, add a new menu entry for the delete function and then attach an event-handler function for the menu's command event in the view class. Edit this function, adding the same code as in the navigation and add record functions and calling the `Delete` function in the document class. Now, add a new member function to the document class. Specify the new function's type as `void`, the declaration as `Delete`, and the access as public. Edit this function, adding the code in Listing 15.13.

LISTING 15.13. THE `CDbAdoDoc` `Delete` FUNCTION.

```
 1: void CDbAdoDoc::Delete()
 2: {
 3:     try
 4:     {
 5:         // Update the current record
 6:         m_piAdoRecordBinding->Update(&m_rsRecSet);
 7:         // Can we delete a record?
 8:         if (m_pRs->Supports(adDelete))
 9:         {
10:             // Make sure the user wants to delete this record
11:             if (AfxMessageBox("Are you sure you want to delete this
                ➥record?",
12:                 MB_YESNO ¦ MB_ICONQUESTION) == IDYES)
13:             {
14:                 // Delete the record
15:                 m_pRs->Delete(adAffectCurrent);
16:                 // Move to the previous record
17:                 m_pRs->MovePrevious();
18:             }
19:         }
20:     }
21:     // Any errors?
22:     catch (_com_error &e)
23:     {
24:         // Generate an error message
25:         GenerateError(e.Error(), e.Description());
26:     }
27: }
```

When you compile and run your application, you should be able to delete any records from the set that you want.

Summary

Today, you learned about Microsoft's newest database access technology, ActiveX Data Objects. You saw how you can use ADO as a simple ActiveX control to provide database access through data-bound controls without any additional programming. You also learned how to import the DLL, providing a rich set of data access functionality that you can use and control in your applications. You learned how to retrieve a set of data, manipulate the records in the set, and save your changes back in the database. You learned two different ways of accessing and updating the data values in a record in the record set and how you can do a little more work up front to save a large amount of work in the midst of the application coding.

Q&A

Q Because Visual C++ doesn't support ADO with its wizards, why would I want to use it?

A ADO is the database access technology direction for Microsoft. It's still in the early stages of this technology, but it will gradually become the data access technology for use with all programming languages and applications.

Q If ADO uses ODBC to get to my database, why wouldn't I want to just go straight to the ODBC interface to access my database?

A ADO can use ODBC to access those databases that don't have a native OLE DB interface. If you are using either Microsoft's SQL Server database or an Oracle database, there are OLE DB interfaces available, in which case ADO would not go through ODBC to get to the database. In these cases, using ADO gives your application better performance than using the ODBC interface. With the upcoming operating system releases from Microsoft, you'll find that using ADO is likely to provide you with access capabilities that extend far beyond conventional databases. ADO is a new technology that you'll start seeing in more use in the coming years. Because of its growing importance, it's a good thing to start working with ADO now so that you'll already be prepared to work with it when it's everywhere.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What does ADO stand for?
2. What does ADO use for database access?
3. What are the objects in ADO?
4. How do you initialize the COM environment?
5. How do you associate a `Connection` object with a `Command` object?
6. How do you associate a `Command` object with and populate a `Recordset` object?

Exercise

Enable and disable the navigation menus and toolbar buttons based on whether the recordset is at the beginning of file (`BOF`) or end of file (`EOF`, renamed to `EndOfFile`).

WEEK 3

DAY 16

Creating Your Own Classes and Modules

Sometimes you need to build a set of application functionality that will be used in an application that another programmer is working on. Maybe the functionality will be used in a number of applications. Another possibility is that you want to separate some functionality from the rest of the application for organizational purposes. You might develop this separate set of functionality and then give a copy of the code to your friend to include in his application, but then every time you make any changes to your set of functionality, it has to be reincorporated into the other set of application code. It would be much more practical if you could give a compiled version of your functionality to the other programmer so that every time you updated your part, all you had to hand over was a new compiled file. The new file could just replace the previous version, without having to make any changes to the other programmer's code.

Well, it is possible to place your set of functionality into a self-contained compiled file, link it into another programmer's application, and avoid adding any new files to the finished application. Today, you will learn

- How to design your own classes.
- How to create compiled modules that can be linked into other applications.
- How to include these modules into an application.

Designing Classes

You've already designed and built your own classes over the past few days, so the basics of creating a new class is not a new topic. Why did you create these classes? Each of the new classes that you created encapsulated a set of functionality that acted as a self-contained unit. These units consisted of both data and functionality that worked together to define the object.

Encapsulation

Object-oriented software design is the practice of designing software in the same way that everything else in the world is designed. For instance, you can consider your car built from a collection of objects: the engine, the body, the suspension, and so on. Each of these objects consists of many other objects. For instance, the engine contains either the carburetor or the fuel injectors, the combustion chamber and pistons, the starter, the alternator, the drive chain, and so on. Once again, each of these objects consists of even more objects.

Each of these objects has a function that it performs. Each of these objects knows how to perform its own functions with little, if any, knowledge of how the other objects perform their functions. Each of the objects knows how it interacts with the other objects and how they are connected to the other objects, but that's about all they know about the other objects. How each of these objects work internally is hidden from the other objects. The brakes on your car don't know anything about how the transmission works, but if you've got an automatic transmission, the brakes do know how to tell the transmission that they are being applied, and the transmission decides how to react to this information.

You need to approach designing new classes for your applications in the same way. The rest of the application objects do not need to know how your objects work; they only need to know how to interact with your objects. This principle, called *encapsulation*, is one of the basic principles of object-oriented software.

Inheritance

Another key principle of object-oriented software design is the concept of *inheritance*. An object can be inherited from another object. The descendent object inherits all the existing functionality of the base object. This allows you to define the descendent object in terms of how it's different from the base object.

Let's look at how this could work with a thermostat. Suppose you had a basic thermostat that you could use in just about any setting. You could set the temperature for it to maintain, and it would turn on the heating or the air-conditioning as needed to maintain that temperature. Now let's say you needed to create a thermostat for use in a freezer. You could start from scratch and build a customized thermostat, or you could take your existing thermostat and specify how the freezer version differs from the original. These differences might include that it's limited to turning on the air conditioning and could never turn on the heater. You would probably also put a strict limit on the range of temperatures to which the thermostat could be set, such as around and below 32° Fahrenheit, or 0° Celsius. Likewise, if you needed a thermostat for an office building, you would probably want to limit the temperature range to what is normally comfortable for people and not allow the temperature to be set to an extremely cold or hot setting.

With inheritance in creating your own classes, this method just described represents the same principle that you want to apply. If possible, you should start with an existing C++ class that has the basic functionality that you need and then program how your class is different from the base class that you inherited from. You have the ability to add new data elements, extend existing functionality, or override existing functionality, as you see fit.

Visual C++ Class Types

In most application projects, when you are creating a new class, you have a few options on the type of class that you are creating. These options are

- Generic class
- MFC class
- Form class

Which of these types of classes you choose to create depends on your needs and what your class will be doing. It also depends on whether your class needs to descend from any of the MFC classes.

Generic Class

You use a generic class for creating a class that is inherited from a class you have already created. This class type is intended for creating classes that are not inherited from any MFC classes (although you have already seen where you need to use it to create classes that are based on MFC classes). If you want to create a more specialized version of the `CLine` class, for instance, a `CRedLine` class, that only drew in red, you create it as a generic class because it's inherited from another class that you created.

When you create a generic class, the New Class Wizard tries to locate the declaration of the base class (the header file with the class declared). If it cannot find the appropriate header file, it tells you that you might need to make sure that the header file with the base class definition is included in the project. If the base class happens to be an MFC class that is not accessible as an MFC class (such as `CObject`), then you can ignore this warning because the correct header file is already part of the project.

MFC Class

If you want to make a reusable class that is based on an existing MFC class, such as an edit box that automatically formats numbers as currency, you want to create an MFC class. The MFC class type is for creating new classes that are inherited from existing MFC classes.

Form Class

The form class is a specialized type of MFC class. You need to create this type of class if you are creating a new form style window. It can be a dialog, form view, or database view class. This new class will be associated with a document class for use with the view class. If you are building a database application, you will probably create a number of this style of classes.

Creating Library Modules

When you create new classes for your application, they might be usable in other applications as well. Often, with a little thought and effort, classes you create can be made flexible enough so that they could be used in other applications. When this is the case, you need some way of packaging the classes for other applications without having to hand over all your source code. This is the issue that library modules address. They allow you to compile your classes and modules into a compiled object code library that can be linked into any other Visual C++ application.

Library modules were one of the first means available to provide compiled code to other programmers for use in their applications. The code is combined with the rest of the application code by the linker as the final step in the compilation process. Library modules are still a viable means of sharing modules with other developers. All the developer needs is the library (`.lib`) file and the appropriate header files that show all the exposed classes, methods, functions, and variables, which the other programmer can access and use. The easiest way to do this is to provide the same header file that you used to create the library file, but you can also edit the header so that only the parts that other programmers need are included.

By using library files to share your modules with other programmers, you are arranging that your part of the application is included in the same executable file as the rest of the application. Your modules are not included in a separate file, such as a DLL or ActiveX control. This results in one less file to be distributed with the application. It also means that if you make any changes to the module, fix any bugs, or enhance any functionality, then the applications that use your module must be relinked. Using library files has a slight disadvantage to creating DLLs, where you may be able to just distribute the new DLL without having to make any changes to the application, but you'll learn all about that tomorrow.

Using Library Modules

To get a good idea of how to use library modules, it's helpful to create a library module, use it in another application, and then make some modifications to the library module. For today's sample application, you'll create a module that generates a random drawing on the window space specified. It'll be able to save and restore any of these drawings. You'll then use this module in an SDI application, where every time a new document is specified, a new drawing is generated. The initial module will only use eight colors and will generate only a limited number of line sequences. Later, you'll modify the module so that it will generate any number of colors and will generate a larger number of line sequences.

Creating the Library Module

To create a library module project, you need to specify in the New dialog that you want to create a Win32 Static Library, as shown in Figure 16.1. This tells Visual C++ that the output from the project compilation will be a library module instead of an executable application. From there, all you have to do is define the classes and add the code. You have the options of including support for MFC and using precompiled headers in your project, as in Figure 16.2, the only step in the Project Wizard.

The library that you will create for today's sample application will consist of two classes. The first class will be the `CLine` class that you first created on Day 10, "Creating Single Document Interface Applications." The second class will be the class that creates the random drawings on the drawing surface. This class will contain an object array of the `CLine` objects that it will create and populate with each of the drawing efforts. This second class will also need functionality to save and restore the drawing, as well as to delete the existing drawing so that a new drawing can be started. It will need to know the dimensions of the drawing area so that it can generate a drawing that will fit in the drawing area. Once you create this module, you'll take a look at how you can use this module in an application project.

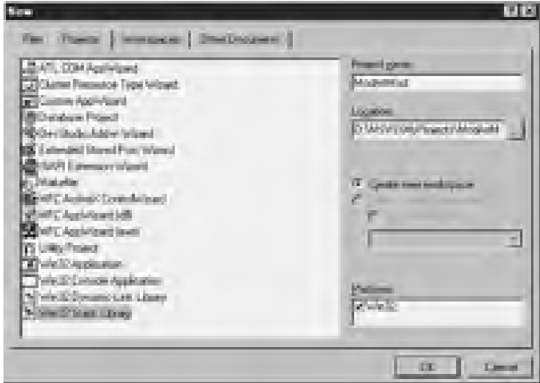


FIGURE 16.1.
Specifying a library module project.

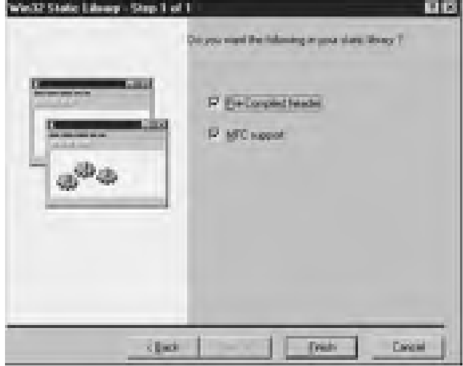


FIGURE 16.2.
Specifying project support options.

Creating a Library Project

To start the library project for today's example, you need to create a new project, specifying that the project is a Win32 Static Library project. Give the project a suitable name and click OK to create the project.

For today's sample project, specify on the one wizard step to include both MFC and precompiled header support. Although the precompiled header support is not necessary, it will speed up most compiles that you perform while building the module.

Once you create your module project, you'll find yourself working with a project that has no classes. You've got a blank slate from which you can create whatever type of module you need.

For your sample project, because you already have the `CLine` class built, copy it from the Day 10 project area into the project directory for today's project. Add both the header

and source code file to today's project by choosing Project|Add To Project\Files. Once you add both of these files to the project, you should see the CLine class appear in the Class View of your project.

Defining the Classes

Now that you've got a basic library module project ready to go, it's time to begin adding the meat of the module. Using the CLine class is an easy way of reusing some functionality that you created earlier in another setting. However, the real functionality of this module will be in its ability to generate random drawings, or squiggles. For this functionality, you'll need to create a new class.

To start this new class, add a new class to the project by selecting New Class from the pop-up menu in the Class View tab. The first thing that you'll notice in the New Class dialog is that you are limited to creating generic classes. Because you are creating a static library that will be linked into the application, Visual C++ is making some assumptions about the type of class that you want to create. Because this is not an MFC project, even though MFC support is included, you are prevented from creating a new MFC or form class. If you need to inherit a new class from an MFC class, you have to add it as if it were a generic class.

Use the New Class dialog to create your new class. Give the class a name that reflects its functionality, such as CModArt, and specify that it's derived from the CObject class as public. You'll receive the same warning that the base class header file cannot be found, but because you specified that MFC support should be included, you can ignore that message.

Once you create your class, you need to add a couple of variables to the class. First, you need somewhere to hold all the lines that will make up the drawing, so you'll add an object array. Second, you need to know the area of the drawing surface, so you'll want a CRect to hold the drawing area specification. You can add both of these variables to your new class using the types and names in Table 16.1.

TABLE 16.1. CModArt VARIABLES.

Type	*Name*	*Access*
static const COLORREF	m_crColors[8]	Public
CRect	m_rDrawArea	Private
CObArray	m_oaLines	Private

Setting the Drawing Area

Before you can draw anything, you need to know the area that you have to draw within. You can add a public function to your class that will copy the passed in `CRect` to the member `CRect` variable. To add this function to your project, add a new member function to your new class, specifying the type as void, the declaration as `SetRect(CRect rDrawArea)`, and the access as public. Edit the function as in Listing 16.1.

LISTING 16.1. THE `CModArt` `SetRect` FUNCTION.

```
1: void CModArt::SetRect(CRect rDrawArea)
2: {
3:     // Set the drawing area rectangle
4:     m_rDrawArea = rDrawArea;
5: }
```

Creating a New Drawing

One of the key pieces to this module is the ability to generate random squiggles that appear on the drawing area. By generating a whole series of these squiggles, your module will be able to create an entire drawing. Starting with the single squiggle, you can design a function that generates one squiggle and then calls this function a number of times to generate the entire drawing.

This first function, the squiggle generator, needs to determine how many lines will be in the squiggle. It needs to determine the color and width of the pen to be used when drawing the squiggle. It also needs to determine the starting point for the squiggle. From this point, it could loop through the appropriate number of lines, generating a new destination to continue the squiggle from the previous destination point.

To add this functionality to your project, add a new member function to the drawing class. Specify the function type as void, the definition as `NewLine`, and the access as private because this function will only be called by the master loop that is determining how many of these squiggles will be in the final drawing. Edit the new function with the code in Listing 16.2.

LISTING 16.2. THE `CModArt` `NewLine` FUNCTION.

```
1:  void CModArt::NewLine()
2:  {
3:      int lNumLines;
4:      int lCurLine;
5:      int nCurColor;
6:      UINT nCurWidth;
```

```
7:       CPoint pTo;
8:       CPoint pFrom;
9:
10:      // Normalize the rectangle before determining the width and height
11:      m_rDrawArea.NormalizeRect();
12:      // get the area width and height
13:      int lWidth = m_rDrawArea.Width();
14:      int lHeight = m_rDrawArea.Height();
15:
16:      // Determine the number of parts to this squiggle
17:      lNumLines = rand() % 100;
18:      // Are there any parts to this squiggle?
19:      if (lNumLines > 0)
20:      {
21:          // Determine the color
22:          nCurColor = rand() % 8;
23:          // Determine the pen width
24:          nCurWidth = (rand() % 8) + 1;
25:          // Determine the starting point for the squiggle
26:          pFrom.x = (rand() % lWidth) + m_rDrawArea.left;
27:          pFrom.y = (rand() % lHeight) + m_rDrawArea.top;
28:          // Loop through the number of segments
29:          for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
30:          {
31:              // Determine the end point of the segment
32:              pTo.x = ((rand() % 20) - 10) + pFrom.x;
33:              pTo.y = ((rand() % 20) - 10) + pFrom.y;
34:              // Create a new CLine object
35:              CLine *pLine = new CLine(pFrom, pTo, nCurWidth,
                         ➥m_crColors[nCurColor]);
36:              try
37:              {
38:                  // Add the new line to the object array
39:                  m_oaLines.Add(pLine);
40:              }
41:              // Did we run into a memory exception?
42:              catch (CMemoryException* perr)
43:              {
44:                  // Display a message for the user, giving him the
45:                  // bad news
46:                  AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
47:                  // Did we create a line object?
48:                  if (pLine)
49:                  {
50:                      // Delete it
51:                      delete pLine;
52:                      pLine = NULL;
53:                  }
54:                  // Delete the exception object
```

16

continues

LISTING 16.2. CONTINUED

```
55:                 perr->Delete();
56:             }
57:             // Set the starting point to the end point
58:             pFrom = pTo;
59:         }
60:     }
61: }
```

In this function, the first thing that you did was get the area that you had available for drawing with the following three lines:

```
m_rDrawArea.NormalizeRect();
int lWidth = m_rDrawArea.Width();
int lHeight = m_rDrawArea.Height();
```

In the first of these lines, you normalized the rectangle. This is necessary to guarantee that the width and height returned in the next two lines are both positive values. Because of the coordinate system used in Windows, getting the width by subtracting the left-side position from the right-side position can result in a negative number. The same can happen with the height. By normalizing the rectangle, you are guaranteeing that you'll get positive results for these two values.

Once you determined the drawing area, you determined the number of line segments you would use in this squiggle:

```
lNumLines = rand() % 100;
```

The `rand` function is capable of returning numbers in a wide range. By getting the modulus of 100, you are guaranteeing that the resulting number will be between 0 and 100. This is a common technique for generating random numbers within a certain range, using the modulus function with the upper limit of the value range (or the upper limit minus the lower limit, if the lower limit is not equal to 0, and then adding the lower limit to the resulting number). You use the same technique to determine the color, width, and starting position for the squiggle:

```
nCurColor = rand() % 8;
nCurWidth = (rand() % 8) + 1;
pFrom.x = (rand() % lWidth) + m_rDrawArea.left;
pFrom.y = (rand() % lHeight) + m_rDrawArea.top;
```

Notice how when you were determining the starting position, you added the left and top of the drawing area to the position that you generated. This guarantees that the starting

position is within the drawing area. Once you enter the loop, generating all the line segments in the squiggle, you limit the available area for the next destination within 10 of the current position:

```
pTo.x = ((rand() % 20) - 10) + pFrom.x;
pTo.y = ((rand() % 20) - 10) + pFrom.y;
CLine *pLine = new CLine(pFrom, pTo, nCurWidth, m_crColors[nCurColor]);
m_oaLines.Add(pLine);
```


You can easily increase this distance to make the drawings more angular. Once you generate the next line segment, you create the line object and add it to the object array. Finally, you set the starting position to the ending position of the line segment you just generated:

```
pFrom = pTo;
```

Now you are ready to go through the loop again and generate the next line segment, until you have generated all line segments in this squiggle.

Now that you can generate a single squiggle, the rest of the process is easy. First, you determine how many squiggles will be in the drawing. Next, you loop for the number of squiggles that need to be generated and call the `NewLine` function once for each squiggle. To add this functionality to your project, add a new member function to the drawing class. Specify the type as void, the declaration as `NewDrawing`, and the access as public. Edit the function as in Listing 16.3.

LISTING 16.3. THE `CModArt NewDrawing` FUNCTION.

```
1:  void CModArt::NewDrawing()
2:  {
3:      int lNumLines;
4:      int lCurLine;
5:
6:      // Determine how many lines to create
7:      lNumLines = rand() % 10;
8:      // Are there any lines to create?
9:      if (lNumLines > 0)
10:     {
11:         // Loop through the number of lines
12:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
13:         {
14:             // Create the new line
15:             NewLine();
16:         }
17:     }
18: }
```

Displaying the Drawing

To draw the set of squiggles on the drawing area, you can add a function that will loop through the object array, calling the `Draw` function on each line segment in the array. This function needs to receive the device context as the only argument and must pass it along to each of the line segments. To add this function to your project, add a new member function to the drawing class. Specify the function type as void, the function declaration as `Draw(CDC *pDC)`, and the access as public. Edit the function as in Listing 16.4.

LISTING 16.4. THE `CModArt` `Draw` FUNCTION.

```
1:  void CModArt::Draw(CDC *pDC)
2:  {
3:      // Get the number of lines in the object array
4:      int liCount = m_oaLines.GetSize();
5:      int liPos;
6:
7:      // Are there any objects in the array?
8:      if (liCount)
9:      {
10:         // Loop through the array, drawing each object
11:         for (liPos = 0; liPos < liCount; liPos++)
12:             ((CLine*)m_oaLines[liPos])->Draw(pDC);
13:     }
14: }
```

Serializing the Drawing

Because you are using the line segment class that you created earlier and have already made serializable, you do not need to add the serialization macros to the drawing class. What you do need to add is a `Serialize` function that passes the archive object on to the object array, letting the object array and line segment objects do all the serialization work. To add this function to your project, add a new member function to the drawing class. Specify the function type as void, the declaration as `Serialize(CArchive &ar)`, and the access as public. Edit the function as in Listing 16.5.

LISTING 16.5. THE `CModArt` `Serialize` FUNCTION.

```
1: void CModArt::Serialize(CArchive &ar)
2: {
3:     // Pass the archive object on to the array
4:     m_oaLines.Serialize(ar);
5: }
```

Clearing the Drawing

To provide full functionality, you need to be able to delete a drawing from the drawing class so that a new drawing can be created or an existing drawing can be loaded. This is a simple matter of looping through the object array and destroying every line segment object and then resetting the object array. To add this functionality to your project, add a new member function to the drawing class. Specify the type as void, the declaration as `ClearDrawing`, and the access as public. Edit the function as in Listing 16.6.

LISTING 16.6. THE `CModArt` `ClearDrawing` FUNCTION.

```
1:  void CModArt::ClearDrawing()
2:  {
3:      // Get the number of lines in the object array
4:      int liCount = m_oaLines.GetSize();
5:      int liPos;
6:
7:      // Are there any objects in the array?
8:      if (liCount)
9:      {
10:         // Loop through the array, deleting each object
11:         for (liPos = 0; liPos < liCount; liPos++)
12:             delete m_oaLines[liPos];
13:         // Reset the array
14:         m_oaLines.RemoveAll();
15:     }
16: }
```

Completing the Class

Finally, to wrap up your drawing class, you need to initialize the random number generator. The random number generator function, `rand`, generates a statistically random number sequence based on a series of mathematical calculations. If the number generator starts with the same number each time, then the sequence of numbers is the same each time. To get the random number generator to produce a different sequence of numbers each time your application runs, you need to seed it with a value that is different each time. The typical way to do this is to feed the current system time into the `srand` function, which seeds the random number generator with a different time each time that the application runs. This seeding of the number generator must be done only once each time the application is run, so you can add this functionality by editing the drawing class constructor with the code in Listing 16.7.

LISTING 16.7. THE `CModArt` CONSTRUCTOR.

```
1: CModArt::CModArt()
2: {
3:     // Initialize the random number generator
4:     srand((unsigned)time(NULL));
5: }
```

To complete the class, you need to include all of the necessary header files for the functionality that you've added to this class. The random number generator needs the `stdlib.h` and `time.h` header files, and the object array needs the header file for the `CLine` class. You also need to populate the color table for use when generating squiggles. You can add all of these finishing touches by scrolling to the top of the source code file for the drawing class and adding lines 5, 6, 9, and 12 through 21 in Listing 16.8.

LISTING 16.8. THE `CModArt` INCLUDES AND COLOR TABLE.

```
1:  // ModArt.cpp: implementation of the CModArt class.
2:  //
3:  //////////////////////////////////////////////////////////////////////
4:
5:  #include <stdlib.h>
6:  #include <time.h>
7:
8:  #include "stdafx.h"
9:  #include "Line.h"
10: #include "ModArt.h"
11:
12: const COLORREF CModArt::m_crColors[8] = {
13:     RGB(   0,   0,   0),    // Black
14:     RGB(   0,   0, 255),    // Blue
15:     RGB(   0, 255,   0),    // Green
16:     RGB(   0, 255, 255),    // Cyan
17:     RGB( 255,   0,   0),    // Red
18:     RGB( 255,   0, 255),    // Magenta
19:     RGB( 255, 255,   0),    // Yellow
20:     RGB( 255, 255, 255)     // White
21: };
```

You have now completed your library module. Before you go any further, you need to compile your project. Once you compile your project, you cannot run anything because you need to create an application that uses your library module in order to run and test your code. To get ready for creating this test application, close the entire workspace so that you will start with a clean workspace for the test application.

Creating a Test Application

To be able to test your module, you need to create a test application that uses the module. This plain application can contain just enough functionality to thoroughly test the module. All you want to do at this point is test all the functionality in the module; you don't have to create a full-blown application.

When you create your test application, you need to include the header file for the drawing class in the relevant classes in your application. In a typical SDI or MDI application, this means including the header file in the document class at a minimum and probably the view and application class source files also. You also have to add the library file that your module created in the application project so that it will be linked into your application.

Creating the Test App Shell

Creating a test application shell is a simple matter of creating a standard SDI or MDI application shell. For the purposes of keeping the test application as simple as possible, it's probably advisable to use an SDI application. However, if you've got some functionality in your module that is intended for use in an MDI application, then that application style might be a better selection as your test application.

For the test application for the sample module you created, create a standard SDI application shell using the AppWizard. Give the project a name such as `TestApp` or some other suitable name. Specify a file extension on the advanced button on the fourth AppWizard step. Otherwise, just go ahead and use the default settings for everything else.

Once you create the application shell, you need to add the library module to the project. You can do this by selecting Project|Add To Project|Files. Once in the Insert Files dialog, specify the file types as library files, as shown in Figure 16.3. Navigate to the debug directory of the module project to find the library module that you created with the previous project. This typically requires moving up one directory level, finding the project directory for the module, and then navigating through it to the debug directory. (If you are building the release version of the module and application, you want to navigate down to the release directory of the module project.) You should be able to find the library file for the module you created, as shown in Figure 16.4. Select this module and click OK to add it to the project.

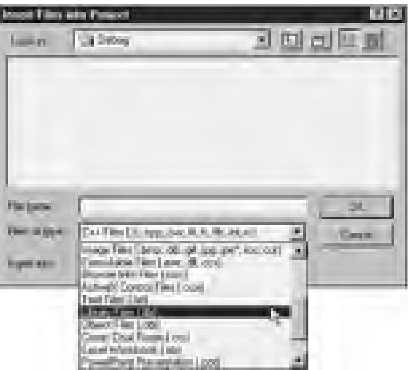


FIGURE 16.3.
Specifying library files.

FIGURE 16.4.
Adding a library file to the project.

Once you add the library file to the project, you also need to add the header files for any of the classes in the module that will be used into the appropriate application source code files. For the test application that you are building, this entails adding line 7 in Listing 16.9. You want to add the same line in the include sections of the source code files for the view and application classes as well.

LISTING 16.9. THE `CTestAppDoc` INCLUDES.

```
1: // TestAppDoc.cpp : implementation of the CTestAppDoc class
2: //
3:
4: #include "stdafx.h"
5: #include "TestApp.h"
6:
7: #include "..\ModArtMod\ModArt.h"
8: #include "TestAppDoc.h"
```

The last thing that you need to do in preparing the application shell is add a variable for any classes from the library module that need to be included in any of the application

classes. In the case of the test application that you are building, this is a variable in the document class of the drawing class that you created in the library module project. To add this variable to your application, add a new member variable to the document class. Specify the variable type as the drawing class from the library module (in this instance, `CModArt`) and specify the name as `m_maDrawing` and the access as private.

Creating a New Drawing

The first place where you want to put some of the functionality of your module is when you are creating a new document. This is the time to be generating a new drawing. As a result, you want to do two things. First, get the drawing area of the view class, passing it along to the drawing object. Second, tell the drawing object to generate a new drawing. This is all fairly straightforward. To add this functionality to your application, edit the `OnNewDocument` function in the document class, adding the lines 9–23 in Listing 16.10.

LISTING 16.10. THE `CTestAppDoc OnNewDocument` FUNCTION.

```
1:  BOOL CTestAppDoc::OnNewDocument()
2:  {
3:      if (!CDocument::OnNewDocument())
4:          return FALSE;
5:
6:      // TODO: add reinitialization code here
7:      // (SDI documents will reuse this document)
8:
9:      // Get the position of the view
10:     POSITION pos = GetFirstViewPosition();
11:     // Did we get a valid position?
12:     if (pos != NULL)
13:     {
14:         // Get a pointer to the view
15:         CView* pView = GetNextView(pos);
16:         RECT lWndRect;
17:         // Get the display area rectangle
18:         pView->GetClientRect(&lWndRect);
19:         // Set the drawing area
20:         m_maDrawing.SetRect(lWndRect);
21:         // Create a new drawing
22:         m_maDrawing.NewDrawing();
23:     }
24:
25:     return TRUE;
26: }
```

Saving and Deleting a Drawing

The other functionality that you want to add to the document class is to save and restore the drawing and to delete the current drawing. These tasks are the last of the document-related functionality of your library module.

To add the functionality to save and restore drawings to your application, edit the `Serialize` function in the document class. Delete all the current contents of the function, replacing it with a call to the drawing object's `Serialize` function, as in Listing 16.11.

LISTING 16.11. THE `CTestAppDoc` `Serialize` FUNCTION.

```
1: void CTestAppDoc::Serialize(CArchive& ar)
2: {
3:     // Serialize the drawing
4:     m_maDrawing.Serialize(ar);
5: }
```

To add the functionality to delete the current drawing so that a new drawing can be generated or a saved drawing can be loaded, you need to add the event handler for the `DeleteContents` function to the document class. In this function, you call the drawing object's `ClearDrawing` function. To add this functionality to your application, use the Class Wizard to add the event handler for the `DeleteContents` event to the document class. Edit this function, adding line 5 in Listing 16.12.

LISTING 16.12. THE `CTestAppDoc` `DeleteContents` FUNCTION.

```
1: void CTestAppDoc::DeleteContents()
2: {
3:     // TODO: Add your specialized code here and/or call the base class
4:     // Delete the drawing
5:     m_maDrawing.ClearDrawing();
6:
7:     CDocument::DeleteContents();
8: }
```

Viewing a Drawing

You need to add one final set of functionality to your test application before you can test your library module: the drawing functionality to the application. This functionality belongs in the view class because it is the object that knows when it needs to redraw itself. Before you can add this functionality to the view class, you need some way for the view class to get access to the drawing object. The easiest way to add this capability is to

add another function to the document class that can be called to get a pointer to the drawing object. Once the view has this pointer, it can call the drawing object's own `Draw` function.

To add the capability to get a pointer to the drawing object to your document class, add a new member function to the document class. Specify the function type as a pointer to the drawing object, in this case, `CModArt*`, and specify the function declaration as `GetDrawing` and the access as public. Edit the function, adding the code in Listing 16.13.

LISTING 16.13. THE `CTestAppDoc GetDrawing` FUNCTION.

```
1: CModArt* CTestAppDoc::GetDrawing()
2: {
3:     // Return the drawing object
4:     return &m_maDrawing;
5: }
```

Adding the drawing functionality to the view class is a simple matter of editing the `OnDraw` function in the view class. In this function, you need to get a pointer to the drawing object and then call its `Draw` function, as in Listing 16.14.

LISTING 16.14. THE `CTestAppView OnDraw` FUNCTION.

```
1:  void CTestAppView::OnDraw(CDC* pDC)
2:  {
3:      CModTestAppDoc* pDoc = GetDocument();
4:      ASSERT_VALID(pDoc);
5:
6:      // TODO: add draw code for native data here
7:
8:      // Get the drawing object
9:      CModArt* m_maDrawing = pDoc->GetDrawing();
10:     // Draw the drawing
11:     m_maDrawing->Draw(pDC);
12: }
```

Once you add all this functionality, you can compile and run your application to test the functionality of your library module. Each time you select File|New from your application menu, a new drawing is created, as in Figure 16.5.

FIGURE 16.5.
Creating random squiggle drawings.

Updating the Library Module

Now that you have a working application, let's go back to the library module and make some changes. Whenever you make any changes to the library module code, no matter how minor, you need to relink all applications that use the module in order to get the updates into those applications. This is because the library module is linked into the EXE of the application. It does not remain in a separate file.

To see how this works, reopen the library module project. You will make three changes to this module. First, you'll increase the number of squiggles that may be included in a single drawing. Second, you'll increase the number of line segments that may make up a single squiggle. Third, you'll generate random colors, beyond just the eight colors included in the color table. Once you make these changes, you'll recompile your library module. Once you generate a new module, you'll relink your test application so that you can incorporate these changes into the application.

To make the first change in your module, increasing the number of squiggles that can be in a drawing, edit the `NewDrawing` function in the drawing class, increasing the modulus value in line 7 of the function, as in Listing 16.15. This will increase the number of possible squiggles in a single drawing from a maximum of 10 to a maximum of 50. There may still be an occasional drawing that doesn't have any squiggles, but you can ignore this possibility for now.

LISTING 16.15. THE MODIFIED `CModArt NewDrawing` FUNCTION.

```
1:  void CModArt::NewDrawing()
2:  {
3:      int lNumLines;
4:      int lCurLine;
```

```
5:
6:     // Determine how many lines to create
7:     lNumLines = rand() % 50;
8:     // Are there any lines to create?
9:     if (lNumLines > 0)
10:    {
11:        // Loop through the number of lines
12:        for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
13:        {
14:            // Create the new line
15:            NewLine();
16:        }
17:    }
18: }
```


With the increased number of squiggles that can be included in a drawing, next you want to increase the number of line segments that may be in a squiggle. To do this, edit the `NewLine` function and increase the modulus number on line 20 in Listing 16.16 from 100 to 200. While you're in this function, you can also increase the number of colors that may be generated for use in each drawing. First, add three integer variable declarations, one for each of the three additive colors (red, green, and blue, as in lines 9 through 11 in Listing 16.16). Next, generate random values for each of these integers between the values of 0 and 255 (lines 26 through 28). Finally, when creating the `CLine` object, pass these colors through the `RGB` function to create the actual color that will be used in the drawing, as in line 41 of Listing 16.16.

LISTING 16.16. THE MODIFIED `CModArt` `NewLine` FUNCTION.

```
1:  void CModArt::NewLine()
2:  {
3:      int lNumLines;
4:      int lCurLine;
5:  //  int nCurColor;
6:      UINT nCurWidth;
7:      CPoint pTo;
8:      CPoint pFrom;
9:      int cRed;
10:     int cBlue;
11:     int cGreen;
12:
13:     // Normalize the rectangle before determining the width and height
14:     m_rDrawArea.NormalizeRect();
15:     // get the area width and height
16:     int lWidth = m_rDrawArea.Width();
```

continues

LISTING 16.16. CONTINUED

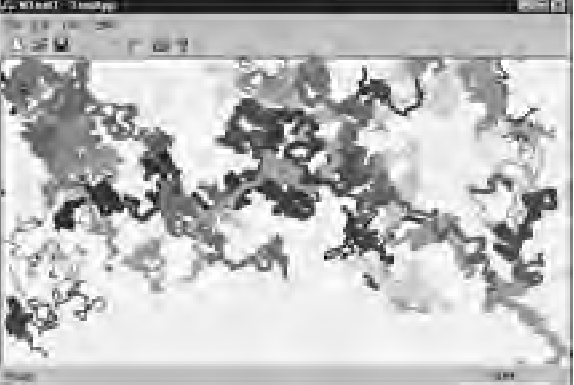
```
17:     int lHeight = m_rDrawArea.Height();
18:
19:     // Determine the number of parts to this squiggle
20:     lNumLines = rand() % 200;
21:     // Are there any parts to this squiggle?
22:     if (lNumLines > 0)
23:     {
24:         // Determine the color
25: //      nCurColor = rand() % 8;
26:         cRed = rand() % 256;
27:         cBlue = rand() % 256;
28:         cGreen = rand() % 256;
29:         // Determine the pen width
30:         nCurWidth = (rand() % 8) + 1;
31:         // Determine the starting point for the squiggle
32:         pFrom.x = (rand() % lWidth) + m_rDrawArea.left;
33:         pFrom.y = (rand() % lHeight) + m_rDrawArea.top;
34:         // Loop through the number of segments
35:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
36:         {
37:             // Determine the end point of the segment
38:             pTo.x = ((rand() % 20) - 10) + pFrom.x;
39:             pTo.y = ((rand() % 20) - 10) + pFrom.y;
40:             // Create a new CLine object
41:             CLine *pLine = new CLine(pFrom, pTo, nCurWidth,
                        ➥RGB(cRed, cGreen, cBlue));
42:             try
43:             {
44:                 // Add the new line to the object array
45:                 m_oaLines.Add(pLine);
46:             }
47:             // Did we run into a memory exception?
48:             catch (CMemoryException* perr)
49:             {
50:                 // Display a message for the user, giving him the
51:                 // bad news
52:                 AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
53:                 // Did we create a line object?
54:                 if (pLine)
55:                 {
56:                     // Delete it
57:                     delete pLine;
58:                     pLine = NULL;
59:                 }
60:                 // Delete the exception object
61:                 perr->Delete();
62:             }
63:             // Set the starting point to the end point
```

```
64:             pFrom = pTo;
65:         }
66:     }
67: }
```

Now that you've made all the necessary changes to the library module, compile it so that it's ready for use in the test application. If you run your test application from the Start | Run Taskbar option, as in Figure 16.6, you'll notice that there is no noticeable difference in how your application behaves. This is because the application hasn't changed. The application is still using the old version of your library module. To get the test application to use the new version of the library module, reopen the test application project in Visual C++. Build the project, which should not do anything other than relink the project, and then run the application. You should see a significant difference in the drawings that your application is now generating, as shown in Figure 16.7.

FIGURE 16.6.
Run the test application from the Start menu.

FIGURE 16.7.
The updated test application.



Summary

Today you learned about how to approach creating and designing new classes for your applications. You learned the differences between the different types of classes that are available to you through the New Class Wizard in Visual C++. You also learned how you can create a library module with a set of your functionality that you can hand to other

programmers for including in their applications. You learned how this module will be linked into the actual applications, thus not requiring a separate file to be distributed along with the applications.

Tomorrow you will learn about a different approach to creating reusable packaged functionality that you can give to other programmers. You will learn how to create DLLs using Visual C++, what the differences are between creating library modules and DLL, and how you need to approach each task.

Q&A

Q Isn't most functionality packaged in DLLs now? Why would I want to create library modules instead of DLLs?

A Yes, the trend toward packaging functionality modules has been to create DLLs instead of library modules for a number of years now. However, there are still instances where library modules are preferable. If you are creating a module that contains proprietary functionality that you do not want to risk exposing to others, but that is needed for any applications that you or another programmer in your company is building, then you would probably want all that functionality packaged in a library module so that it is internal to the application. Using library modules makes it effectively inaccessible to your competition without significant disassembly and reverse engineering efforts.

Q Why does the header file need to be included in the application that is using my library file?

A The application needs to know about the objects that are in the library file. In the sample application, you didn't need to include the header file for the `CLine` class because the application didn't directly use or reference the `CLine` class. However, the application did use the drawing object that was in your library module, so it did need to know about that object, how it is defined, and what functions are available for it. If you don't want the other programmers to know all of the internal structure of your classes, then you can create another header file to be distributed with your library module. This header would contain definitions of all of the same classes that are in the library module but would only provide the public functions and variables that the other programmers can actually access.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. When do you want to create a new MFC class?
2. When you make changes to a library file, what do you have to do to the applications that use the library file?
3. What are the different types of classes that you can create?
4. When you package some functionality in a library file, what do you need to give to other programmers who want to use your library module?
5. What are two of the basic principles in object-oriented software design?

Exercise

Separate the `CLine` class into a different library module from the drawing class so that you have two library modules instead of one. Link them into the test application.

WEEK 3

DAY 17

Sharing Your Functionality with Other Applications—Creating DLLs

Yesterday you learned how you could create a set of functionality that might be useful for multiple applications and how you could package it in a library file that could be linked into those applications. Today you will learn how to do this same thing, only with a much more dynamic package.

Often, a family of applications will have some functionality in common. When you place this shared functionality into DLLs instead of library modules, all the applications can use the same functionality with only a single copy of the functionality distributed in the form of DLLs, instead of duplicating the same functionality in each of the applications. This method saves disk space on any systems where the applications are installed.

Today, you will learn

- About the different types of DLLs that you can create with Visual C++ and how to determine which type best suits your needs.
- How to build two of these types of DLLs and the different approaches for the various DLL types.
- How to use the functionality for both of these types of DLLs in a Visual C++ application.
- How to determine when an application needs to be relinked when you make modifications to a DLL that is used by the application.

Why Create DLLs?

Dynamic link libraries (DLL) were introduced by Microsoft back in the early days of Windows. DLLs are similar to library modules in that they both contain sets of functionality that have been packaged for use by applications. The difference is when the applications link to the library. With a library module (LIB), the application is linked to the functionality in the library during the compile and build process. The functionality contained in the library file becomes part of the application executable file. With a DLL, the application links to the functionality in the library file when the application is run. The library file remains a separate file that is referenced and called by the application.

There are several reasons for creating DLLs instead of library module files. First, you can reduce the size of the application executable files by placing functionality that is used by multiple applications into DLLs that are shared by all of the applications. You can update and modify functionality in the DLLs without having to update the application executable (assuming that the exported interface for the DLL doesn't change). Finally, you can use DLLs with just about any other Windows programming language, which makes your functionality available to a wider number of programmers, not just fellow Visual C++ programmers.

Creating and Using DLLs

DLLs are library files with compiled code that can be used by other applications. The DLLs expose certain functions and classes to these applications by exporting the function. When a function is exported, it is added to a table that is included in the DLL. This table lists the location of all exported functions contained in the DLL, and it is used to locate and call each of these functions. Any functions that are not exported are not added to this table, and they cannot be seen or called by any outside application or DLL.

An application can call the functions in the DLL in two ways. The more involved method of calling these functions is to look up the location of the desired function in the DLL and get a pointer to this function. The pointer can then be used to call the function.

The other, much easier way (and the only way that you'll use in any of the examples in this book) is to link the application with the LIB file that is created with the DLL. This LIB file is treated by the linker as a standard library file, just like the one that you created yesterday. However, this LIB file contains stubs for each of the exported functions in the DLL. A stub is a pseudo-function that has the same name and argument list as the real function. In the interior of the function stub is a small amount of code that calls the real function in the DLL, passing all of the arguments that were passed to the stub. This allows you to treat the functions in the DLL as if they were part of the application code and not as a separate file.

Note

> The LIB file for a DLL is automatically created for the DLL during the compiling of the DLL. There is nothing extra that you need to do to create it.

> Not only is it easier to create your applications using the LIB files for any DLLs that you will be using, but also it can be safer when running the application. When you use the LIB files, any DLLs that are used by your application are loaded into memory the moment the application is started. If any of the DLLs are missing, the user is automatically informed of the problem by Windows, and your application does not run. If you don't use the LIB files, then you are responsible for loading the DLL into memory and handling any errors that occur if the DLL cannot be found.

There are two types of DLLs that you can easily create using Visual C++. These two types are MFC extension DLLs and regular DLLs.

> You can create other types of DLLs using Visual C++. All these other types of DLLs involve a significant amount of ActiveX functionality, so they are beyond the scope of this book. If you need to build ActiveX in-process server DLLs, or other types of ActiveX DLLs, I recommend that you find an advanced book on Visual C++ that provides significant coverage for these topics.

MFC Extension DLLs

MFC DLLs are the easiest to code and create because you can treat them just like any other collection of classes. For any classes that you want to export from the DLL, the only thing that you need to add is the `AFX_EXT_CLASS` macro in the class declaration, as follows:

```
class AFX_EXT_CLASS CMyClass
{
.
.
.
};
```

This macro exports the class, making it accessible to Visual C++ applications. You need to include this macro in the header file that is used by the applications that will use the DLL, where it will import the class from the DLL so that it can be used.

The one drawback to creating MFC extension DLLs is that they cannot be used by any other programming languages. They can be used with other C++ compilers as long as the compiler supports MFC (such as with Borland's and Symantec's C++ compilers).

Regular DLLs

The other type of DLL is a regular DLL. This type of DLL exports standard functions from the DLL, not C++ classes. As a result, this type of DLL can require a little more thought and planning than an MFC extension DLL. Once inside the DLL, you can use classes all you want, but you must provide straight function calls to the external applications.

To export a function, declare it as an export function by preceding the function name with

```
extern "C" <function type> PASCAL EXPORT <function declaration>
```

Include all this additional stuff in both the header file function prototype and the actual source code. The `extern "C"` portion declares that this is a standard C function call so that the C++ name mangler does not mangle the function name. `PASCAL` tells the compiler that all function arguments are to be passed in `PASCAL` order, which places the arguments on the stack in the reverse order from how they are normally placed. Finally, `EXPORT` tells the compiler that this function is to be exported from the DLL and can be called outside the DLL.

The other thing that you need to do to export the functions from your DLL is to add all the exported function names to the DEF file for the DLL project. This file is used to build the stub LIB file and the export table in the DLL. It contains the name of the DLL,

or library, a brief description of the DLL, and the names of all functions that are to be exported. This file has to follow a specific format, so you should not modify the default DEF file that is automatically created by the DLL Wizard other than to add exported function names. A typical DEF file follows:

```
LIBRARY     "mydll"
DESCRIPTION 'mydll Windows Dynamic Link Library'

EXPORTS
    ; Explicit exports can go here
    MyFunc1
    MyFunc2
```

If you are using MFC classes in your regular DLLs, you need to call the `AFX_MANAGE_STATE` macro as the first line of code in all exported functions. This is necessary to make the exported functions threadsafe, which allows your class functions to be called simultaneously by two or more programs (or threads). The `AFX_MANAGE_STATE` macro takes a single argument, a pointer to a `AFX_MODULE_STATE` structure, which can be retrieved by calling the `AfxGetStaticModuleState` function. A typical exported function that uses MFC looks like the following:


```
extern "C" void PASCAL EXPORT MyFunc(...)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    // normal function body here
.
.
.
}
```

Designing DLLs

When you are designing your DLLs, you should be aware that any of the functions in your DLLs can be called simultaneously by multiple applications all running at the same time. As a result, all the functionality in any DLLs that you create must be threadsafe.

All variables that hold any values beyond each individual function call must be held and maintained by the application and not the DLL. Any application variables that must be manipulated by the DLL must be passed in to the DLL as one of the function arguments. Any global variables that are manipulated within the DLL may be swapped with variables from other application processes while the function is running, leading to unpredictable results.

Creating and Using an MFC Extension DLL

To see how easy it is to create and use an MFC extension DLL, you'll convert the library module that you created yesterday into an MFC extension DLL today. After you see how easy it is, and what types of changes you have to make to use the DLL, you'll then reimplement the same functionality as a regular DLL so that you can get an understanding of the different approaches that are necessary with the two DLL styles.

Creating the MFC Extension DLL

To convert the library module you created yesterday into an MFC extension DLL, you need to create a new MFC DLL Wizard project, specifying that the project is an MFC extension DLL. Copy the source code and header files for the line and drawing classes into the project directory. Load the files for the line and drawing classes into the current project. Add the `AFX_EXT_CLASS` macro to the drawing class. Finally, move the color table from a global static table to a local variable inside the function that creates the squiggles.

To create this DLL, start a new project. Give the project a suitable name, such as `ModArtDll`, and specify that the project is an MFC AppWizard (DLL) project, as in Figure 17.1. Once in the DLL Wizard, specify that the DLL is an MFC Extension DLL, as in Figure 17.2.

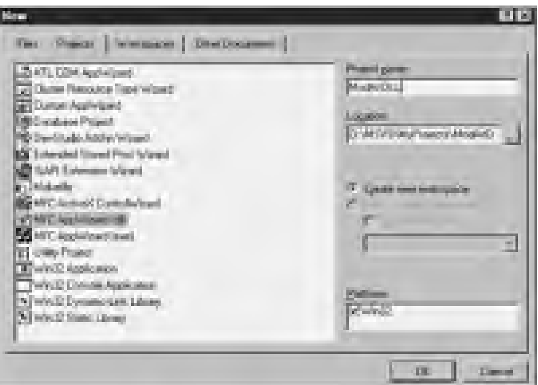


FIGURE 17.1.
Selecting the MFC DLL Wizard.

Once you create the DLL shell, open the file explorer and copy the source code and header files for the line and drawing classes (`line.cpp`, `line.h`, `ModArt.cpp`, and `ModArt.h`) from the library module project you created yesterday into the project directory that you just created. Add all four of these files to the project. Both classes should appear in the Class View of the workspace pane.

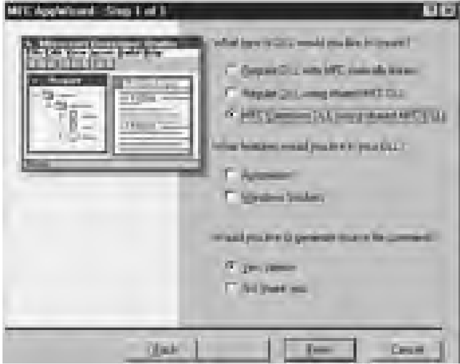


FIGURE 17.2.
Specifying the DLL type.

Open the header file containing the definition of the drawing class. Add the AFX_EXT_CLASS macro to the class declaration as shown in Listing 17.1. Remove the color table variable from the class declaration also.

LISTING 17.1. THE MODIFIED CModArt CLASS DECLARATION.

```
1:  class AFX_EXT_CLASS CModArt : public CObject
2:  {
3:  public:
4:      void NewDrawing();
5:      virtual void Serialize(CArchive &ar);
6:      void Draw(CDC *pDC);
7:      void ClearDrawing();
8:      void SetRect(CRect rDrawArea);
9:      CModArt();
10:     virtual ~CModArt();
11:
12: private:
13:     void NewLine();
14:     CRect m_rDrawArea;
15:     CObArray m_oaLines;
16: };
```

You cannot have public static tables in DLLs, so you cannot declare the color table as a public, static member of the drawing class, as it was yesterday. As a result, you'll move it to a local variable in the NewLine member function. Edit the NewLine function to add this local variable and to reset the function to behave as it did in its initial incarnation, as in Listing 17.2.

LISTING 17.2. THE CModArt NewLine FUNCTION.

```
1:  void CModArt::NewLine()
2:  {
3:      int lNumLines;
4:      int lCurLine;
5:      int nCurColor;
6:      UINT nCurWidth;
7:      CPoint pTo;
8:      CPoint pFrom;
9:
10:     // Normalize the rectangle before determining the width and height
11:     m_rDrawArea.NormalizeRect();
12:     // get the area width and height
13:     int lWidth = m_rDrawArea.Width();
14:     int lHeight = m_rDrawArea.Height();
15:
16:     COLORREF crColors[8] = {
17:     RGB(   0,   0,   0),    // Black
18:     RGB(   0,   0, 255),    // Blue
19:     RGB(   0, 255,   0),    // Green
20:     RGB(   0, 255, 255),    // Cyan
21:     RGB( 255,   0,   0),    // Red
22:     RGB( 255,   0, 255),    // Magenta
23:     RGB( 255, 255,   0),    // Yellow
24:     RGB( 255, 255, 255)     // White
25:     };
26:
27:     // Determine the number of parts to this squiggle
28:     lNumLines = rand() % 100;
29:     // Are there any parts to this squiggle?
30:     if (lNumLines > 0)
31:     {
32:         // Determine the color
33:         nCurColor = rand() % 8;
34:         // Determine the pen width
35:         nCurWidth = (rand() % 8) + 1;
36:         // Determine the starting point for the squiggle
37:         pFrom.x = (rand() % lWidth) + m_rDrawArea.left;
38:         pFrom.y = (rand() % lHeight) + m_rDrawArea.top;
39:         // Loop through the number of segments
40:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
41:         {
42:             // Determine the end point of the segment
43:             pTo.x = ((rand() % 20) - 10) + pFrom.x;
44:             pTo.y = ((rand() % 20) - 10) + pFrom.y;
45:             // Create a new CLine object
46:             CLine *pLine = new CLine(pFrom, pTo, nCurWidth,
                        ➥crColors[nCurColor]);
47:             try
```

```
48:             {
49: // Add the new line to the object array
50:                 m_oaLines.Add(pLine);
51:             }
52:             // Did we run into a memory exception?
53:             catch (CMemoryException* perr)
54:             {
55:                 // Display a message for the user, giving him the
56:                 // bad news
57:                 AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
58:                 // Did we create a line object?
59:                 if (pLine)
60:                 {
61:                     // Delete it
62:                     delete pLine;
63:                     pLine = NULL;
64:                 }
65:                 // Delete the exception object
66:                 perr->Delete();
67:             }
68:             // Set the starting point to the end point
69:             pFrom = pTo;
70:         }
71:     }
72: }
```


After making these changes to the drawing class, you are ready to compile your DLL. Once you compile the DLL, switch over to the file explorer, find the DLL in the debug subdirectory under the project directory, and copy the DLL to the debug directory in the test application project directory.

Adapting the Test Application

To adapt the test application to use the DLL, open the test application project that you created yesterday. You are going to delete the library module that you created yesterday and add the LIB file that was created with the DLL. You are also going to change the header file that is included for the drawing class. After making these two changes, your test application will be ready to use with the DLL.

To delete the library module from the project, open the File View in the workspace pane. Select the LIB file from the list of project files and press the Delete key. Once you delete the library file from the project, select Project|Add To Project|Files from the main menu. Specify the Library Files (.lib) file type, and then navigate to the debug directory of the DLL project. Select the LIB file that was created with your DLL, in this case, `ModArtDll.lib`. Click OK to add the file to the project.

Once you add the DLL's LIB file, edit the source-code files for the document, view, and application classes, changing the include of the drawing class to point to the project directory of the DLL, as in line 7 in Listing 17.3.

LISTING 17.3. THE `CTestAppDoc` INCLUDES.

```
1: // TestAppDoc.cpp : implementation of the CTestAppDoc class
2: //
3:
4: #include "stdafx.h"
5: #include "TestApp.h"
6:
7: #include "..\ModArtDll\ModArt.h"
8: #include "TestAppDoc.h"
```

After making this change to all three source-code files, you are ready to compile and run your test application. You should find your test application running just like it did yesterday, only generating shorter squiggles and using only the eight colors in the color table.

Changing the DLL

Now that you have the test application running with the DLL, you'll make the same changes to the DLL that you made to the library module yesterday. You'll increase the number of squiggles that can be included in a drawing, increase the possible length of each squiggle, and generate any number of colors for use in the squiggles.

To make these changes, switch back to the DLL project. Increase the number of lines that may be generated in the `NewDrawing` member function of the drawing class. Increase the possible length of the squiggles in the `NewLine` member function, and add the random colors back in, as in Listing 17.4.

LISTING 17.4. THE MODIFIED `CModArt NewLine` FUNCTION.

```
1:  void CModArt::NewLine()
2:  {
3:      int lNumLines;
4:      int lCurLine;
5:  //  int nCurColor;
6:      UINT nCurWidth;
7:      CPoint pTo;
8:      CPoint pFrom;
9:      int cRed;
10:     int cBlue;
11:     int cGreen;
12:
```

```
13:     // Normalize the rectangle before determining the width and height
14:     m_rDrawArea.NormalizeRect();
15:     // get the area width and height
16:     int lWidth = m_rDrawArea.Width();
17:     int lHeight = m_rDrawArea.Height();
18:
19: //    COLORREF crColors[8] = {
20: //    RGB(   0,   0,   0),    // Black
21: //    RGB(   0,   0, 255),    // Blue
22: //    RGB(   0, 255,   0),    // Green
23: //    RGB(   0, 255, 255),    // Cyan
24: //    RGB( 255,   0,   0),    // Red
25: //    RGB( 255,   0, 255),    // Magenta
26: //    RGB( 255, 255,   0),    // Yellow
27: //    RGB( 255, 255, 255)     // White
28: //    };
29:
30:     // Determine the number of parts to this squiggle
31:     lNumLines = rand() % 200;
32:     // Are there any parts to this squiggle?
33:     if (lNumLines > 0)
34:     {
35:         // Determine the color
36: //      nCurColor = rand() % 8;
37:         cRed = rand() % 256;
38:         cBlue = rand() % 256;
39:         cGreen = rand() % 256;
40:         // Determine the pen width
41:         nCurWidth = (rand() % 8) + 1;
42:         // Determine the starting point for the squiggle
43:         pFrom.x = (rand() % lWidth) + m_rDrawArea.left;
44:         pFrom.y = (rand() % lHeight) + m_rDrawArea.top;
45:         // Loop through the number of segments
46:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
47:         {
48:             // Determine the end point of the segment
49:             pTo.x = ((rand() % 20) - 10) + pFrom.x;
50:             pTo.y = ((rand() % 20) - 10) + pFrom.y;
51:             // Create a new CLine object
52:             CLine *pLine = new CLine(pFrom, pTo, nCurWidth,
                        ➥RGB(cRed, cGreen, cBlue));
53:             try
54:             {
55:                 // Add the new line to the object array
56:                 m_oaLines.Add(pLine);
57:             }
58:             // Did we run into a memory exception?
59:             catch (CMemoryException* perr)
60:             {
```

17

continues

LISTING 17.4. CONTINUED

```
61:                     // Display a message for the user, giving him the
62:                     // bad news
63:                     AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
64:                     // Did we create a line object?
65:                     if (pLine)
66:                     {
67:                         // Delete it
68:                         delete pLine;
69:                         pLine = NULL;
70:                     }
71:                     // Delete the exception object
72:                     perr->Delete();
73:                 }
74:                 // Set the starting point to the end point
75:                 pFrom = pTo;
76:             }
77:         }
78: }
```

After making these changes, compile the DLL again. Once you compile the DLL, switch to the file explorer and copy the DLL into the debug directory of the test application again. Once you copy the DLL, run the test application from the Start|Run Taskbar, as in Figure 17.3. You should find that the application has been updated, and it is now including more squiggles and using many different colors.

FIGURE 17.3.
Starting the sample application.

Creating and Using a Regular DLL

You might think that you broke the rules about using variables that are not owned by the application in a DLL when you created and used the MFC extension DLL. Well, you didn't. The instance of the drawing class was a member of the document class in the test application. It was created and maintained by the application, not the DLL. Now that you are turning your attention to implementing the same functionality as a regular DLL, this will become clearer.

To convert the MFC extension DLL into a regular DLL, you'll have to convert the drawing class into a series of regular function calls. In the course of making this conversion,

the object array must become a member variable of the application document class and must be passed as an argument to every exported function in the DLL.

Creating the Regular DLL

To convert the MFC extension DLL into a regular DLL, you have to start a new project. Visual C++ has to build a project that tells the compiler what type of file it's creating. You can create this new project using the same steps you used to create the MFC extension DLL project, but specify on the DLL Wizard that you are creating a regular DLL. (You can leave the wizard at the default settings.) Once you create the project, you can copy the line and drawing class source code and header files into the project directory and add these files to the project. Once you add these files to the project, you need to begin the process of converting the drawing class into a series of straight function calls.

Altering the Header File

To start with, you need to radically alter the header file for the drawing class so that it will work for a regular DLL. You have to eliminate every trace of the actual class from the header file, leaving only the function calls. All of these functions must be passed in any objects that they need to work with. (Every function will need to be passed the object array as one of its arguments.) Next, you need to slightly modify all the function names so that the compiler does not get mixed up and call a member function of any class by mistake (such as the `Serialize` function). Finally, each of the public functions must be declared as an exportable function. Making these changes to the header file, you end up replacing the entire class declaration with the function prototypes in Listing 17.5.

LISTING 17.5. THE MODIFIED `ModArt` HEADER FILE.

```
1: extern "C" void PASCAL EXPORT ModArtNewDrawing(CRect pRect,
       ➥CObArray *poaLines);
2: extern "C" void PASCAL EXPORT ModArtSerialize(CArchive &ar,
       ➥CObArray *poaLines);
3: extern "C" void PASCAL EXPORT ModArtDraw(CDC *pDC, CObArray *poaLines);
4: extern "C" void PASCAL EXPORT ModArtClearDrawing(CObArray *poaLines);
5: void NewLine(CRect pRect, CObArray *poaLines);
```

Notice that the object array is always passed as a pointer to each of these functions. Because these functions are adding and removing objects from the array, they need to work with the actual array and not a copy of it.

Adapting the Drawing Generation Functions

Moving to the source-code file, you need to make numerous small yet significant changes to these functions. Starting with the `NewDrawing` function, you need to pass in the `CRect` object to get the drawing area. You dropped the function for setting the drawing area because you have no local variables in which you can hold this object. As a result, you are better off passing it to the drawing generation functions. The other change is where you pass in the object array as another argument to the function. You aren't doing anything with either of these arguments in this function, just passing them along to the squiggle generating function. The other alteration in this function is the addition of the `AFX_MANAGE_STATE` macro as the first line in the body of the function. After making these changes, the `NewDrawing` function will look like the one in Listing 17.6.

LISTING 17.6. THE `ModArtNewDrawing` FUNCTION.

```
1:  extern "C" void PASCAL EXPORT ModArtNewDrawing(CRect pRect,
        ➥CObArray *poaLines)
2:  {
3:      AFX_MANAGE_STATE(AfxGetStaticModuleState());
4:      // normal function body here
5:      int lNumLines;
6:      int lCurLine;
7:
8:      // Initialize the random number generator
9:      srand((unsigned)time(NULL));
10:     // Determine how many lines to create
11:     lNumLines = rand() % 50;
12:     // Are there any lines to create?
13:     if (lNumLines > 0)
14:     {
15:         // Loop through the number of lines
16:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
17:         {
18:             // Create the new line
19:             NewLine(pRect, poaLines);
20:         }
21:     }
22: }
```

Another change that is required in the `NewDrawing` function is the addition of the random number generator seeding on line 9. Because there is no class constructor any more, you cannot seed the random number generator in it. Therefore, the next logical place to add this is in the `NewDrawing` function before any random numbers are generated.

On the NewLine function, the changes are more extensive. First, the CRect object and the object array are passed in as arguments. Second, because this is not an exported function, you do not need to add the AFX_MANAGE_STATE macro. Third, all the places where the CRect member variable is used must be changed to use the CRect that is passed as an argument to the function. Finally, when adding objects to the object array, you need to change this to use the object array pointer that was passed as an argument. Making these changes leaves you with the code in Listing 17.7.

LISTING 17.7. THE NewLine FUNCTION.

```
 1:   void NewLine(CRect pRect, CObArray *poaLines)
 2:   {
 3:       int lNumLines;
 4:       int lCurLine;
 5:   //  int nCurColor;
 6:       UINT nCurWidth;
 7:       CPoint pTo;
 8:       CPoint pFrom;
 9:       int cRed;
10:      int cBlue;
11:      int cGreen;
12:
13:      // Normalize the rectangle before determining the width and
         ➥height
14:      pRect.NormalizeRect();
15:      // get the area width and height
16:      int lWidth = pRect.Width();
17:      int lHeight = pRect.Height();
18:
19:  //    COLORREF crColors[8] = {
20:  //    RGB(   0,   0,   0),    // Black
21:  //    RGB(   0,   0, 255),    // Blue
22:  //    RGB(   0, 255,   0),    // Green
23:  //    RGB(   0, 255, 255),    // Cyan
24:  //    RGB( 255,   0,   0),    // Red
25:  //    RGB( 255,   0, 255),    // Magenta
26:  //    RGB( 255, 255,   0),    // Yellow
27:  //    RGB( 255, 255, 255)     // White
28:  //    };
29:
30:      // Determine the number of parts to this squiggle
31:      lNumLines = rand() % 200;
32:      // Are there any parts to this squiggle?
33:      if (lNumLines > 0)
34:      {
35:          // Determine the color
```

17

continues

LISTING 17.7. CONTINUED

```
36: //      nCurColor = rand() % 8;
37:         cRed = rand() % 256;
38:         cBlue = rand() % 256;
39:         cGreen = rand() % 256;
40:         // Determine the pen width
41:         nCurWidth = (rand() % 8) + 1;
42:         // Determine the starting point for the squiggle
43:         pFrom.x = (rand() % lWidth) + pRect.left;
44:         pFrom.y = (rand() % lHeight) + pRect.top;
45:         // Loop through the number of segments
46:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
47:         {
48:             // Determine the end point of the segment
49:             pTo.x = ((rand() % 20) - 10) + pFrom.x;
50:             pTo.y = ((rand() % 20) - 10) + pFrom.y;
51:             // Create a new CLine object
52:             CLine *pLine = new CLine(pFrom, pTo, nCurWidth,
                        ➥RGB(cRed, cGreen, cBlue));
53:             try
54:             {
55:                 // Add the new line to the object array
56:                 poaLines->Add(pLine);
57:             }
58:             // Did we run into a memory exception?
59:             catch (CMemoryException* perr)
60:             {
61:                 // Display a message for the user, giving him the
62:                 // bad news
63:                 AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
64:                 // Did we create a line object?
65:                 if (pLine)
66:                 {
67:                     // Delete it
68:                     delete pLine;
69:                     pLine = NULL;
70:                 }
71:                 // Delete the exception object
72:                 perr->Delete();
73:             }
74:             // Set the starting point to the end point
75:             pFrom = pTo;
76:         }
77:     }
78: }
```

Adapting the Other Functions

Making the necessary changes to the other functions is less involved than the changes to the drawing generation functions. With the rest of the functions, you must add a pointer

to the object array as a function argument and then alter the uses of the array to use the pointer instead of the no longer existing member variable. You also need to add the AFX_MANAGE_STATE macro as the first line in each of the remaining functions. This leaves you with the functions shown in Listings 17.8, 17.9, and 17.10.

LISTING 17.8. THE ModArtDraw FUNCTION.

```
1:  extern "C" void PASCAL EXPORT ModArtDraw(CDC *pDC, CObArray *poaLines)
2:  {
3:      AFX_MANAGE_STATE(AfxGetStaticModuleState());
4:      // normal function body here
5:      // Get the number of lines in the object array
6:      int liCount = poaLines.GetSize();
7:      int liPos;
8:
9:      // Are there any objects in the array?
10:     if (liCount)
11:     {
12:         // Loop through the array, drawing each object
13:         for (liPos = 0; liPos < liCount; liPos++)
14:             ((CLine*)poaLines[liPos])->Draw(pDC);
15:     }
16: }
```

17

LISTING 17.9. THE ModArtSerialize FUNCTION.

```
1: extern "C" void PASCAL EXPORT ModArtSerialize(CArchive &ar,
       ➥CObArray *poaLines)
2: {
3:     AFX_MANAGE_STATE(AfxGetStaticModuleState());
4:     // normal function body here
5:     // Pass the archive object on to the array
6:     poaLines.Serialize(ar);
7: }
```

LISTING 17.10. THE ModArtClearDrawing FUNCTION.

```
1:  extern "C" void PASCAL EXPORT ModArtClearDrawing(CObArray *poaLines)
2:  {
3:      AFX_MANAGE_STATE(AfxGetStaticModuleState());
4:      // Normal function body here
5:      // Get the number of lines in the object array
6:      int liCount = poaLines.GetSize();
7:      int liPos;
```

continues

LISTING 17.10. CONTINUED

```
8:
9:      // Are there any objects in the array?
10:     if (liCount)
11:     {
12:         // Loop through the array, deleting each object
13:         for (liPos = 0; liPos < liCount; liPos++)
14:             delete poaLines[liPos];
15:         // Reset the array
16:         poaLines.RemoveAll();
17:     }
18: }
```

Once you make the changes to these functions, the only thing remaining is to remove all code for the class constructor and destructor, along with the code for the `SetRect` function.

Building the Module Definition File

Before you compile the DLL, you need to add all the function names to the module definition file. You can find this file in the list of source-code files in the File View of the workspace pane. When you open this file, you'll find that it briefly describes the module that you are building in generic terms. You'll see a place at the bottom of the file where you can add the exports for the DLL. Edit this file, adding the exportable function names, as in Listing 17.11.

LISTING 17.11. THE DLL MODULE DEFINITION FILE.

```
1:  ; ModArtRDll.def : Declares the module parameters for the DLL
2:
3:  LIBRARY         "ModArtRDll"
4:  DESCRIPTION     'ModArtRDll Windows Dynamic Link Library'
5:
6:  EXPORTS
7:       ; Explicit exports can go here
8:       ModArtNewDrawing
9:       ModArtSerialize
10:      ModArtDraw
11:      ModArtClearDrawing
```

You are now ready to compile your regular DLL. Once you compile the DLL, copy it into the debug directory of the test application.

Adapting the Test Application

To adapt the test application to use the new DLL that you have just created, you need to make a number of changes. First, you need to change the member variable of the document class from an instance of the drawing class to the object array. Next, you need to change the include in the document and view source code to include the header from the new DLL instead of the header from the old DLL. (You can completely remove the include in the application source-code file.) Drop the DLL LIB file and add the LIB file for the new DLL to the project. Change all of the drawing class function calls to call functions in the new DLL instead. Finally, change the `GetDrawing` function in the document class so that it returns a pointer to the object array, instead of the drawing object.

You can start making these changes by deleting the LIB file from the test application project. Once you delete the file, add the LIB file for the new DLL to the project by selecting Project|Add To Project|Files from the main menu.

Once you switch the LIB files in the project, edit the source code for the document and view classes to change the include statement, changing the project directory to the new DLL project directory. You can edit the application class source-code file and remove the include from this file. Because you are not creating any instances of the drawing class, the application file doesn't need to know about anything in the DLL.

Once you make all those changes, open the header file for the document class. Edit the document class declaration: Change the function type of the `GetDrawing` function to return a pointer to an object array, remove the drawing class variable, and add an object array variable, as in Listing 17.12. Make only these three changes; do not change anything else in the class declaration.

LISTING 17.12. THE `CTestAppDoc` CLASS DECLARATION.

```
1:  class CTestAppDoc : public CDocument
2:  {
3:  protected: // create from serialization only
4:      CTestAppDoc();
5:      DECLARE_DYNCREATE(CTestAppDoc)
6:  .
7:  .
8:  .
9:  // Implementation
10: public:
11:     CObArray* GetDrawing();
12:     virtual ~CTestAppDoc();
13: .
```

continues

LISTING 17.12. CONTINUED

```
14: .
15: .
16: private:
17:     CObArray m_oaLines;
18: };
```

Modifying the Document Functions

Now that you've made the general changes to the test application, it's time to start making the functionality changes. All the calls to a class method of the drawing object must be changed to the appropriate function call in the new DLL.

The changes necessary in the `OnNewDocument` function consist of dropping the function call to pass the `CRect` to the drawing object and replacing the `NewDocument` function call with the new DLL function—in this instance, `ModArtNewDrawing`, as shown in line 19 in Listing 17.13.

LISTING 17.13. THE `CTestAppDoc OnNewDocument` FUNCTION.

```
1:  BOOL CTestAppDoc::OnNewDocument()
2:  {
3:      if (!CDocument::OnNewDocument())
4:          return FALSE;
5:
6:      // TODO: add reinitialization code here
7:      // (SDI documents will reuse this document)
8:
9:      // Get the position of the view
10:     POSITION pos = GetFirstViewPosition();
11:     // Did we get a valid position?
12:     if (pos != NULL)
13:     {
14:         // Get a pointer to the view
15:         CView* pView = GetNextView(pos);
16:         RECT lWndRect;
17:         // Get the display area rectangle
18:         pView->GetClientRect(&lWndRect);
19:         // Create a new drawing
20:         ModArtNewDrawing(lWndRect, &m_oaLines);
21:     }
22:
23:     return TRUE;
24: }
```

In the `Serialize` function, change the drawing object `Serialize` function call to the new DLL serialization function—in this case, `ModArtSerialize`, as in Listing 17.14.

LISTING 17.14. THE `CTestAppDoc Serialize` FUNCTION.

```
1: void CTestAppDoc::Serialize(CArchive& ar)
2: {
3:     // Serialize the drawing
4:     ModArtSerialize(ar, &m_oaLines);
5: }
```

For the `DeleteContents` function, you need to change the call to the `ClearDrawing` function to the new DLL function, `ModArtClearDrawing`, as in line 5 of Listing 17.15.

LISTING 17.15. THE `CTestAppDoc DeleteContents` FUNCTION.

```
1: void CTestAppDoc::DeleteContents()
2: {
3:     // TODO: Add your specialized code here and/or call the base class
4:     // Delete the drawing
5:     ModArtClearDrawing(&m_oaLines);
6:
7:     CDocument::DeleteContents();
8: }
```

Finally, for the `GetDrawing` function, you need to change the function declaration to designate that it's returning a pointer to an object array, just as you did in the header file. Next, you need to change the variable that is being returned to the object array variable that you added to the header file, as in Listing 17.16.

LISTING 17.16. THE `CTestAppDoc GetDrawing` FUNCTION.

```
1: CObArray* CTestAppDoc::GetDrawing()
2: {
3:     // Return the drawing object
4:     return &m_oaLines;
5: }
```

Modifying the View Functions

Switching to the view class, there's only one simple change to make to the `OnDraw` function. In this function, you need to change the type of pointer retrieved from the `GetDrawing` function from a drawing object to an object array object, as in line 9 of

Listing 17.17. Next, call the DLL function, `ModArtDraw`, to perform the drawing on the window, as shown in line 11.

LISTING 17.17. THE `CTestAppView` `OnDraw` FUNCTION.

```
1:  void CTestAppView::OnDraw(CDC* pDC)
2:  {
3:      CModTestAppDoc* pDoc = GetDocument();
4:      ASSERT_VALID(pDoc);
5:
6:      // TODO: add draw code for native data here
7:
8:      // Get the drawing object
9:      CObArray* m_oaLines = pDoc->GetDrawing();
10:     // Draw the drawing
11:     ModArtDraw(pDC, m_oaLines);
12: }
```

After making all these changes to the test application, you are ready to compile and test it. You should find that the application is working just as it did with the previous DLL. You can also play around with it, going back and changing the DLL, copying the new DLL into the debug directory for the test application, and seeing how the changes are reflected in the behavior of the test application.

The particular example of a regular DLL that you developed in this exercise is still not usable by other programming languages. The reason is that you are passing MFC classes as the arguments for each of the DLL's functions. This still limits the usage to other applications that are built using MFC. To make this DLL truly portable, you need to pass the bare-bones structures instead of the classes (such as the `RECT` structure instead of the `CRect` class) and then convert the structures to the classes inside the DLL.

Summary

Today you learned about two more ways that you can package your functionality for other programmers. You learned how you can easily package your classes as an MFC extension DLL and how easily it can be used by a Visual C++ application. You saw how you can make changes to the DLL without having to recompile the applications that use it. You also learned what's involved in creating a regular DLL that can be used with other, non-Visual C++ applications. You saw how you needed to convert the exported classes from the DLL into standard C-style functions and what's involved in adapting an application to use this style of DLL.

Q&A

Q How can I convert the regular DLL so that it can be used by non-Visual C++ applications?

A First, you have to make all the arguments to the functions use the bare-bones structures, instead of the MFC classes. For instance, to convert the `ModArtNewDrawing` function, change it to receive the `RECT` structure instead of the `CRect` class and also to receive a generic pointer instead of a pointer to an object array. You have to make the conversions to the appropriate classes in the DLL, as in lines 4 through 9 in Listing 17.18.

LISTING 17.18. THE `ModArtNewDrawing` FUNCTION.

```
1:  extern "C" void PASCAL EXPORT ModArtNewDrawing(RECT spRect,
        ➥LPVOID lpoaLines)
2:  {
3:      AFX_MANAGE_STATE(AfxGetStaticModuleState());
4:      CRect pRect;
5:      pRect.top = spRect.top;
6:      pRect.left = spRect.left;
7:      pRect.right = spRect.right;
8:      pRect.bottom = spRect.bottom;
9:      CObArray* poaLines = (CObArray*)lpoaLines;
10:     // Normal function body here
11:     int m_lNumLines;
12:     int m_lCurLine;
13:
14:     // Initialize the random number generator
15:     srand((unsigned)time(NULL));
16:     // Determine how many lines to create
17:     m_lNumLines = rand() % 50;
18:     // Are there any lines to create?
19:     if (m_lNumLines > 0)
20:     {
21:         // Loop through the number of lines
22:         for (m_lCurLine = 0; m_lCurLine < m_lNumLines; m_lCurLine++)
23:         {
24:             // Create the new line
25:             NewLine(pRect, poaLines);
26:         }
27:     }
28: }
```

You also have to add functions to create and destroy the object array, with the application storing the object array as a generic pointer as in Listing 17.19.

LISTING 17.19. THE `ModArtInit` FUNCTION.

```
1: extern "C" LPVOID PASCAL EXPORT ModArtInit()
2: {
3:     AFX_MANAGE_STATE(AfxGetStaticModuleState());
4:     // Create the object array
5:     return (LPVOID)new CObArray;
6: }
```

Q When do I need to recompile the applications that use my DLLs?

A Whenever you change any of the exported function calls. Changing, adding, or removing arguments to any of these functions would mean recompiling the applications that use the DLL. If you are working with an MFC extension DLL, the applications that use the DLL need to be recompiled if the public interface for the exported classes change or a new function or variable is added or removed. It doesn't matter if the application isn't using any of the functions that were changed; it's still good practice to recompile the applications, just to be sure.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What kind of DLL do you have to create to make classes in the DLL available to applications?
2. What do you have to add to the class to export it from a DLL?
3. What kind of DLL can be used with other programming languages?
4. If you make changes in a DLL, do you have to recompile the applications that use the DLL?
5. What function does the LIB file provide for a DLL?

Exercises

1. Separate the line class into its own MFC extension DLL and use it with the second (regular) DLL.
2. Alter the line class DLL so that it uses a consistent line width for all lines.

DAY 18

Doing Multiple Tasks at One Time—Multitasking

Sometimes it is convenient to let your applications do more than one thing at a time. Your application could write a backup file or print in the background while the user is working on the same document. Your application could perform calculations while the user enters new data or draws multiple images simultaneously. There are many different reasons why you might want to add this capability, called multitasking, to your applications. Windows provides several facilities specifically for building this into applications.

Today, you will learn

- How tasks can be performed while an application is idle.
- How tasks can run independently of the rest of the application.
- How to coordinate access to resources that are shared between multiple independent tasks.
- How to start and stop independently running tasks.

What Is Multitasking?

In the days of Windows 3.*x*, all Windows applications were single-threaded, with only one path of execution at any point in time. The version of multitasking that Windows 3.*x* offered is known as cooperative multitasking. The key to cooperative multitasking is that each individual application makes the decision about when to give up the processor for another application to perform any processing that it might be waiting to perform. As a result, Windows 3.*x* was susceptible to an ill-behaved application that would hold other applications prisoner while it performed some long, winding process or even got itself stuck in some sort of loop.

With Windows NT and Windows 95, the nature of the operating system changed. No more cooperative multitasking—the new method was preemptive multitasking. With preemptive multitasking, the operating system decides when to take the processor away from the current application and give the processor to another application that is waiting for it. It doesn't matter whether the application that has the processor is ready to give it up; the operating system takes the processor without the application's permission. This is how the operating system enables multiple applications to perform computation-intensive tasks and still let all the applications make the same amount of progress in each of their tasks. Giving this capability to the operating system prevents a single application from holding other applications prisoner while hogging the processor.

With the 16/32 bit structure of Windows 95, it is still possible for an ill-behaved 16-bit application to lock up the system because a large amount of 16-bit code remains a core part of the operating system. The 16-bit code on Windows 95 is still a cooperative multitasking environment, so only one application can execute 16-bit code at a time. Because all the `USER` functions, and a good portion of the `GDI` functions, thunk down to the 16-bit version, it is still possible for a single 16-bit application to lock up the entire system.

On Windows NT, if all of the 16-bit applications run in a shared memory space, an ill-behaved application can lock up all of the 16-bit applications, but this has no effect on any 32-bit applications.

Performing Multiple Tasks at One Time

Along with the capability to allow multiple applications to run simultaneously comes the capability for a single application to execute multiple threads of execution at any time. A thread is to an application what an application is to the operating system. If an application has multiple threads running, it is basically running multiple applications

within the whole application. This lets the application accomplish more things simultaneously, such as when Microsoft Word checks your spelling at the same time you are typing your document.

Idle Processing Threads

One of the easiest ways to let your application perform multiple tasks at one time is to add some idle processing tasks. An idle processing task is a task that is performed when an application is sitting idle. Literally, a function in the application class is called when there are no messages in the application message queue. The idea behind this function is that while the application is idle, it can perform work such as cleaning up memory (also known as garbage collection) or writing to a print spool.

The `OnIdle` function is a holdover from the Windows 3.*x* days. It is a member of the `CWinApp` class, from which your application class is inherited. By default, no processing in this function is added by the AppWizard, so if you want this function in your application, you must add it to your application class through the Class Wizard. (`OnIdle` is one of the available messages for the `App` class in your applications.)

The `OnIdle` function receives one argument, which is the number of times the `OnIdle` function has been called since the last message was processed by your application. You can use this to determine how long the application has been idle and when to trigger any functionality that you need to run if the application is idle for more than a certain amount of time.

One of the biggest concerns in adding `OnIdle` processing to your applications is that any functionality you add must be small and must quickly return control to the user. When an application performs any `OnIdle` processing, the user cannot interact with the application until the `OnIdle` processing finishes and returns control to the user. If you need to perform some long, drawn-out task in the `OnIdle` function, break it up into many small and quick tasks so that control can return to the user; then, you can continue your `OnIdle` task once the message queue is empty again. This means that you also have to track your application's progress in the `OnIdle` task so that the next time the `OnIdle` function is called, your application can pick up the task where it left off.

Spawning Independent Threads

If you really need to run a long background task that you don't want interfering with the user, you should spawn an independent thread. A thread is like another application running within your application. It does not have to wait until the application is idle to perform its tasks, and it does not cause the user to wait until it takes a break.

The two methods of creating an independent thread use the same function to create and start the thread. To create and start an independent thread, you call the `AfxBeginThread` function. You can choose to pass it a function to call for performing the thread's tasks, or you can pass it a pointer to the runtime class for an object derived from the `CWinThread` class. Both versions of the function return a pointer to a `CWinThread` object, which runs as an independent thread.

In the first version of the `AfxBeginThread` function, the first argument is a pointer to the main function for the thread to be started. This function is the equivalent of the main function in a C/C++ program. It controls the top-level execution for the thread. This function must be defined as a `UINT` function with a single `LPVOID` argument:

```
UINT MyThreadFunction( LPVOID pParam);
```

This version of the `AfxBeginThread` function also requires a second argument that is passed along to the main thread function as the only argument to that function. This argument can be a pointer to a structure containing any information that the thread needs to know to perform its job correctly.

The first argument to the second version of the `AfxBeginThread` function is a pointer to the runtime class of an object derived from the `CWinThread` class. You can get a pointer to the runtime class of your `CWinThread` class by using the `RUNTIME_CLASS` macro, passing your class as the only argument.

After these initial arguments, the rest of the arguments to the `AfxBeginThread` function are the same for both versions, and they are all optional. The first of these arguments is the priority to be assigned to the thread, with a default priority of `THREAD_PRIORITY_NORMAL`. Table 18.1 lists the available thread priorities.

TABLE 18.1. THREAD PRIORITIES.

Priority	*Description*
`0`	The thread will inherit the thread priority of the application creating the thread.
`THREAD_PRIORITY_NORMAL`	A normal (default) priority.
`THREAD_PRIORITY_ABOVE_NORMAL`	1 point above normal priority.
`THREAD_PRIORITY_BELOW_NORMAL`	1 point below normal priority.
`THREAD_PRIORITY_HIGHEST`	2 points above normal priority.
`THREAD_PRIORITY_LOWEST`	2 points below normal priority.
`THREAD_PRIORITY_IDLE`	Priority level of 1 for most threads (all non-real-time threads).
`THREAD_PRIORITY_TIME_CRITICAL`	Priority level of 15 for most threads (all non-real-time threads).

Thread priority controls how much of the CPU's time the thread gets in relation to the other threads and processes running on the computer. If a thread will not be performing any tasks that need to be completed quickly, you should give the thread a lower priority when creating it. It is not advisable to give a thread a priority higher than normal unless it is vitally important that the thread perform its tasks faster than other processes running on the computer. The higher a thread's priority, the more CPU time that thread will receive, and the less CPU time all other processes and threads on the computer will receive.

The next argument to the `AfxBeginThread` function is the stack size to be provided for the new thread. The default value for this argument is `0`, which provides the thread the same size stack as the main application.

The next argument to the `AfxBeginThread` function is the thread-creation flag. This flag can contain one of two values and controls how the thread is started. If `CREATE_SUSPENDED` is passed as this argument, the thread is created in suspended mode. The thread does not run until the `ResumeThread` function is called for the thread. If you supply `0` as this argument, which is the default value, the thread begins executing the moment it is created.

The final argument to the `AfxBeginThread` function is a pointer to the security attributes for the thread. The default value for this argument is `NULL`, which causes the thread to be created with the same security profile as the application. Unless you are building applications to run on Windows NT and you need to provide a thread with a specific security profile, you should always use the default value for this argument.

Building Structures

Imagine that you have an application running two threads, each parsing its own set of variables at the same time. Imagine also that the application is using a global object array to hold these variables. If the method of allocating and resizing the array consisted of checking the current size and adding one position onto the end of the array, your two threads might build an array populated something like the one in Figure 18.1, where array positions populated by the first thread are intermingled with those created by the second thread. This could easily confuse each thread as it retrieves values from the array for its processing needs because each thread is just as likely to pull a value that actually belongs to the other thread. This would cause each thread to operate on wrong data and return the wrong results.

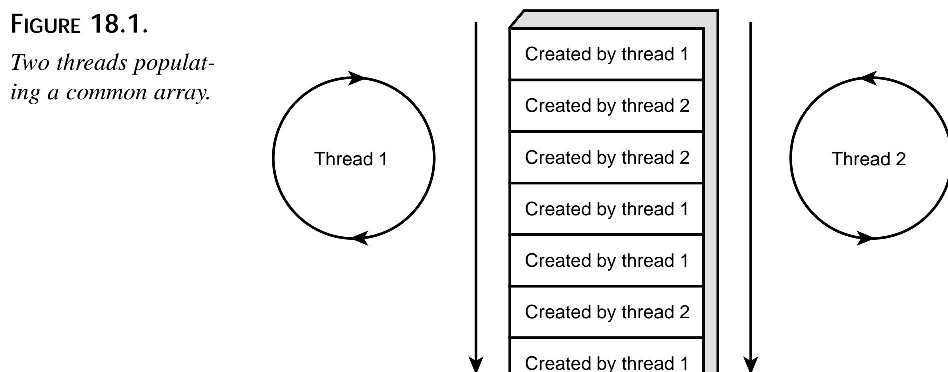


FIGURE 18.1.
Two threads populating a common array.

If the application built these arrays as localized arrays, instead of global arrays, it could keep access to each array limited to only the thread that builds the array. In Figure 18.2, for example, there is no intermingling of data from multiple threads. If you take this approach to using arrays and other memory structures, each thread can perform its processing and return the results to the client, confident that the results are correct because the calculations were performed on uncorrupted data.

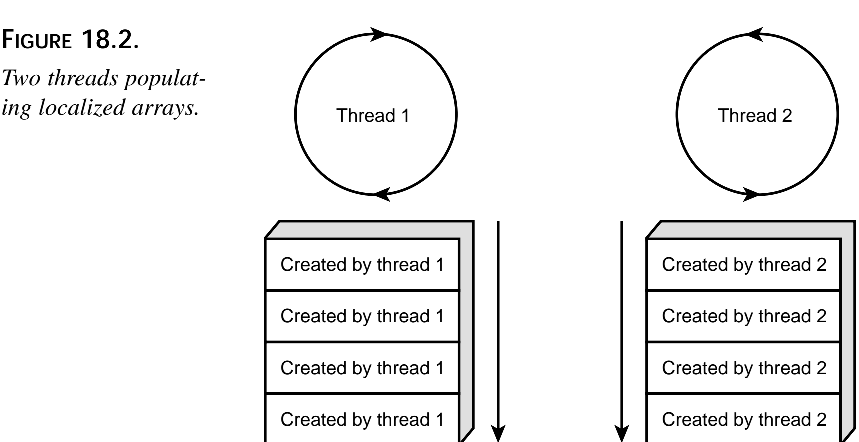


FIGURE 18.2.
Two threads populating localized arrays.

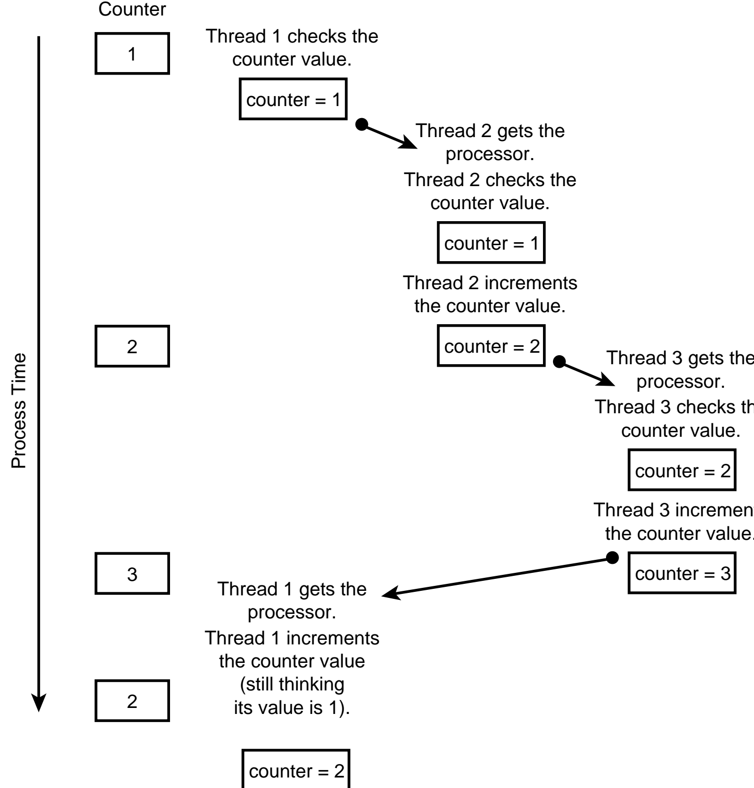
Managing Access to Shared Resources

Not all variables can be localized, and you will often want to share some resources between all the threads running in your applications. Such sharing creates an issue with multithreaded applications. Suppose that three threads all share a single counter, which is generating unique numbers. Because you don't know when control of the processor is

going to switch from one thread to the next, your application might generate duplicate "unique" numbers, as shown in Figure 18.3.

FIGURE 18.3.

Three threads sharing a single counter.



As you can see, this sharing doesn't work too well in a multithreaded application. You need a way to limit access to a common resource to only one thread at a time. In reality, there are four mechanisms for limiting access to common resources and synchronizing processing between threads, all of which work in different ways and whose suitability depends on the circumstances. The four mechanisms are

- Critical sections
- Mutexes
- Semaphores
- Events

Critical Sections

A critical section is a mechanism that limits access to a certain resource to a single thread within an application. A thread enters the critical section before it needs to work with the specific shared resource and then exits the critical section after it is finished accessing the resource. If another thread tries to enter the critical section before the first thread exits the critical section, the second thread is blocked and does not take any processor time until the first thread exits the critical section, allowing the second to enter. You use critical sections to mark sections of code that only one thread should execute at a time. This doesn't prevent the processor from switching from that thread to another; it just prevents two or more threads from entering the same section of code.

If you use a critical section with the counter shown in Figure 18.3, you can force each thread to enter a critical section before checking the current value of the counter. If each thread does not leave the critical section until after it has incremented and updated the counter, you can guarantee that—no matter how many threads are executing and regardless of their execution order—truly unique numbers are generated, as shown in Figure 18.4.

If you need to use a critical section object in your application, create an instance of the `CCriticalSection` class. This object contains two methods, `Lock` and `Unlock`, which you can use to gain and release control of the critical section.

Mutexes

Mutexes work in basically the same way as critical sections, but you use mutexes when you want to share the resource between multiple applications. By using a mutex, you can guarantee that no two threads running in any number of applications will access the same resource at the same time.

Because of their availability across the operating system, mutexes carry much more overhead than critical sections do. A mutex lifetime does not end when the application that created it shuts down. The mutex might still be in use by other applications, so the operating system must track which applications are using a mutex and then destroy the mutex once it is no longer needed. In contrast, critical sections have little overhead because they do not exist outside the application that creates and uses them. After the application ends, the critical section is gone.

If you need to use a mutex in your applications, you will create an instance of the `CMutex` class. The constructor of the `CMutex` class has three available arguments. The first argument is a boolean value that specifies whether the thread creating the `CMutex` object is the initial owner of the mutex. If so, then this thread must release the mutex before any other threads can access it.

FIGURE 18.4.

Three threads using the same counter, which is protected by a critical section.

Counter

Process Time

1

Thread 1 enters the critical section.

Thread 1 checks the counter value.

counter = 1

Thread 2 gets the processor.

Thread 2 tries to enter the critical section and is blocked.

Thread 3 gets the processor.

Thread 3 tries to enter the critical section and is blocked.

Thread 1 gets the processor.

Thread 1 increments the counter value.

2

counter = 2

Thread 1 leaves the critical section.

Thread 3 gets the processor.

Thread 3 enters the critical section.

Thread 3 checks the counter value.

counter = 2

Thread 3 increments the counter value.

3

counter = 3

Thread 3 leaves the critical section.

Thread 2 gets the processor.

Thread 2 enters the critical section.

Thread 2 checks the counter value.

counter = 3

Thread 2 increments the counter value.

4

counter = 4

Thread 2 leaves the critical section.

The second argument is the name for the mutex. All the applications that need to share the mutex can identify it by this textual name. The third and final argument to the `CMutex` constructor is a pointer to the security attributes for the mutex object. If a `NULL` is passed for this pointer, the mutex object uses the security attributes of the thread that created it.

Once you create a `CMutex` object, you can lock and unlock it using the `Lock` and `Unlock` member functions. This allows you to build in the capabilities to control access to a shared resource between multiple threads in multiple applications.

Semaphores

Semaphores work very differently from critical sections and mutexes. You use semaphores with resources that are not limited to a single thread at a time— a resource that should be limited to a fixed number of threads. A semaphore is a form of counter, and threads can increment or decrement it. The trick to semaphores is that they cannot go any lower than zero. Therefore, if a thread is trying to decrement a semaphore that is at zero, that thread is blocked until another thread increments the semaphore.

Suppose you have a queue that is populated by multiple threads, and one thread removes the items from the queue and performs processing on each item. If the queue is empty, the thread that removes and processes items has nothing to do. This thread could go into an idle loop, checking the queue every so often to see whether something has been placed in it. The problem with this scenario is that the thread takes up processing cycles doing absolutely nothing. These processor cycles could go to another thread that does have something to do. If you use a semaphore to control the queue, each thread that places items into the queue can increment the semaphore for each item placed in the queue, and the thread that removes the items can decrement the semaphore just before removing each item from the queue. If the queue is empty, the semaphore is zero, and the thread removing items is blocked on the call to decrement the queue. This thread does not take any processor cycles until one of the other threads increments the semaphore to indicate that it has placed an item in the queue. Then, the thread removing items is immediately unblocked, and it can remove the item that was placed in the queue and begin processing it, as shown in Figure 18.5.

If you need to use a semaphore in your application, you can create an instance of the `CSemaphore` class. This class has four arguments that can be passed to the class constructor. The first argument is the starting usage count for the semaphore. The second argument is the maximum usage count for the semaphore. You can use these two arguments to control how many threads and processes can have access to a shared resource at any one time. The third argument is the name for the semaphore, which is used to identify

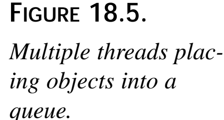
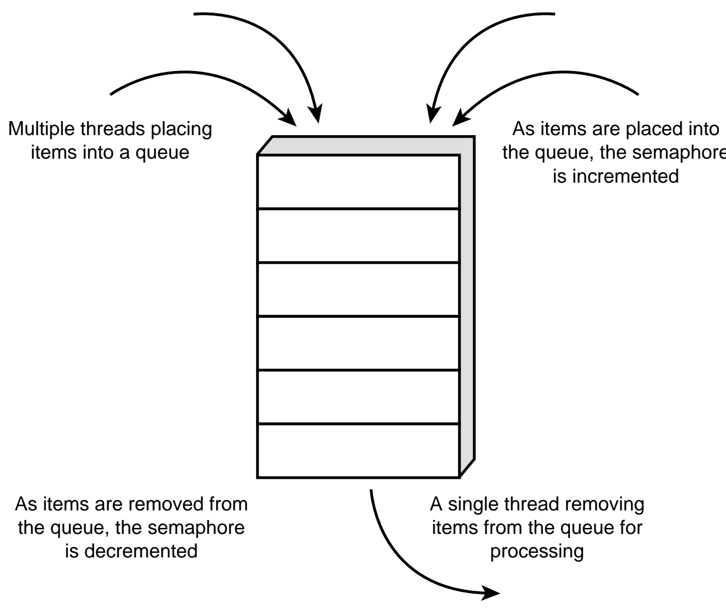


FIGURE 18.5.
Multiple threads placing objects into a queue.



the semaphore by all applications running on the system, just as with the `CMutex` class. The final argument is a pointer to the security attributes for the semaphore.

With the `CSemaphore` object, you can use the `Lock` and `Unlock` member functions to gain or release control of the semaphore. When you call the `Lock` function, if the semaphore usage count is greater than zero, the usage count is decremented and your program is allowed to continue. If the usage count is already zero, the `Lock` function waits until the usage count is incremented so that your process can gain access to the shared resource. When you call the `Unlock` function, the usage count of the semaphore is incremented.

Events

As much as thread synchronization mechanisms are designed to control access to limited resources, they are also intended to prevent threads from using unnecessary processor cycles. The more threads running at one time, the slower each of those threads performs its tasks. Therefore, if a thread does not have anything to do, block it and let it sit idle, allowing other threads to use more processor time and thus run faster until the conditions are met that provide the idle thread with something to do.

This is why you use events—to allow threads to be idle until the conditions are such that they have something to do. Events take their name from the events that drive most Windows applications, only with a twist. Thread synchronization events do not use the normal event queuing and handling mechanisms. Instead of being assigned a number and

then waiting for that number to be passed through the Windows event handler, thread synchronization events are actual objects held in memory. Each thread that needs to wait for an event tells the event that it is waiting for it to be triggered and then goes to sleep. When the event is triggered, it sends wake-up calls to every thread that told it that it was waiting to be triggered. The threads pick up their processing at the exact point where they each told the event that they were waiting for it.

If you need to use an event in your application, you can create a `CEvent` object. You need to create the `CEvent` object when you need to access and wait for the event. Once the `CEvent` constructor has returned, the event has occurred and your thread can continue on its way.

The constructor for the `CEvent` class can take four arguments. The first argument is a boolean flag to indicate whether the thread creating the event will own it initially. This value should be set to `TRUE` if the thread creating the `CEvent` object is the thread that will determine when the event has occurred.

The second argument to the `CEvent` constructor specifies whether the event is an automatic or manual event. A manual event remains in the signaled or unsignaled state until it is specifically set to the other state by the thread that owns the event object. An automatic event remains in the unsignaled state most of the time. When the event is set to the signaled state, and at least one thread has been released and continued on its execution path, the event is returned to the unsignaled state.

The third argument to the event constructor is the name for the event. This name will be used to identify the event by all threads that need to access the event. The fourth and final argument is a pointer to the security attributes for the event object.

The `CEvent` class has several member functions that you can use to control the state of the event. Table 18.2 lists these functions.

TABLE 18.2. `CEvent` MEMBER FUNCTIONS.

Function	*Description*
`SetEvent`	Puts the event into the signaled state.
`PulseEvent`	Puts the event into the signaled state and then resets the event back to the unsignaled state.
`ResetEvent`	Puts the event into the unsignaled state.
`Unlock`	Releases the event object.

Building a Multitasking Application

To see how you can create your own multitasking applications, you'll create an application that has four spinning color wheels, each running on its own thread. Two of the spinners will use the `OnIdle` function, and the other two will run as independent threads. This setup will enable you to see the difference between the two types of threading, as well as learn how you can use each. Your application window will have four check boxes to start and stop each of the threads so that you can see how much load is put on the system as each runs alone or in combination with the others.

Creating a Framework

For the application that you will build today, you'll need an SDI application framework, with the view class inherited from the `CFormView` class, so that you can use the dialog editor to lay out the few controls on the window. It will use the document class to house the spinners and the independent threads, whereas the view will have the check boxes and variables that control whether each thread is running or idle.

To create the framework for your application, create a new project workspace using the MFC Application Wizard. Give your application a suitable project name, such as `Tasking`.

In the AppWizard, specify that you are creating a single document (SDI) application. You can accept the defaults through most of the rest of the AppWizard, although you won't need support for ActiveX controls, a docking toolbar, the initial status bar, or printing and print preview, so you can unselect these options if you so desire. Once you reach the final AppWizard step, specify that your view class is inherited from the `CFormView` class.

Once you create the application framework, remove the static text from the main application window, and add four check boxes at approximately the upper-left corner of each quarter of the window space, as in Figure 18.6. Set the properties of the check boxes as in Table 18.3.

TABLE 18.3. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Check Box	ID	`IDC_CBONIDLE1`
	Caption	`On &Idle Thread 1`
Check Box	ID	`IDC_CBTHREAD1`
	Caption	`Thread &1`

continues

18

TABLE 18.3. CONTINUED

Object	Property	Setting
Check Box	ID	IDC_CBONIDLE2
	Caption	On Idle &Thread 2
Check Box	ID	IDC_CBTHREAD2
	Caption	Thread &2

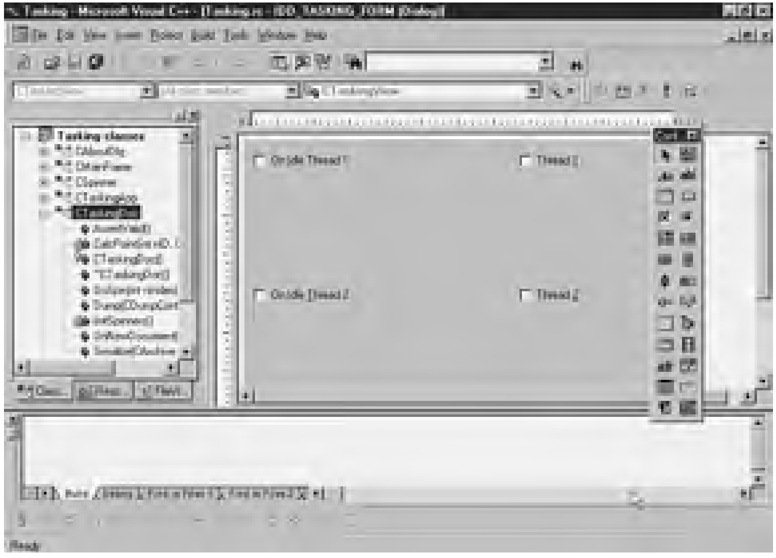


FIGURE 18.6.
The main window design.

Once you add the check boxes to the window and configure their properties, use the Class Wizard to add a variable to each of them. Make all of the variables `BOOL`, and give them names like in Table 18.4.

TABLE 18.4. CONTROL VARIABLES.

Object	Name	Category	Type
IDC_CBONIDLE1	m_bOnIdle1	Value	BOOL
IDC_CBONIDLE2	m_bOnIdle2	Value	BOOL
IDC_CBTHREAD1	m_bThread1	Value	BOOL
IDC_CBTHREAD2	m_bThread2	Value	BOOL

Designing Spinners

Before you can start adding threads to your application, you'll create the spinning color wheel that the threads will operate. Because four of these color wheels will all spin independently of each other, it makes sense to encapsulate all of the functionality into a single class. This class will track what color is being drawn, where in the spinning it needs to draw the next line, the size of the color wheel, and the location of the color wheel on the application window. It will also need a pointer to the view class so that it can get the device context in which it is supposed to draw itself. For the independent spinners, the class will need a pointer to the flag that will control whether the spinner is supposed to be spinning.

To start the spinner class, create a new generic class, inherited from the `CObject` base class. Provide the new class with a name that is descriptive of what it will be doing, such as `CSpinner`.

Setting Spinner Variables

Once you create a new class for your spinner object, you'll add some variables to the class. To follow good object-oriented design principles, you need to make all these variables private and add methods to the class to set and retrieve the values of each.

The variables you'll add are

- The current color.
- The current position in the rotation of the color wheel.
- The size of the color wheel.
- The position on the application window for the color wheel.
- The color table from which the colors are picked for drawing in the color wheel.
- A pointer to the view object so that the spinner can get the device context that it will need for drawing on the window.
- A pointer to the check box variable that specifies whether the thread should be running.

You can add all these variables to the spinner class using the names and types specified in Table 18.5.

TABLE 18.5. `CSpinner` CLASS VARIABLES.

Name	*Type*	*Description*
`m_crColor`	`int`	The current color from the color table.
`m_nMinute`	`int`	The position in the rotation around the wheel.

continues

TABLE 18.5. CONTINUED

Name	*Type*	*Description*
m_iRadius	int	The radius (size) of the wheel.
m_pCenter	CPoint	The center point of the wheel.
m_crColors[8]	static COLORREF	The color table with all of the colors to be drawn in the color wheel.
m_pViewWnd	CWnd*	A pointer to the view object.
m_bContinue	BOOL*	A pointer to the check box variable that specifies whether this thread should be running.

Once you add all the necessary variables, you need to make sure that your class either initializes them or provides a suitable means of setting and retrieving the values of each. All the integer variables can be initialized as zero, and they'll work their way up from that point. The pointers should be initialized with `NULL`. You can do all of this initialization in the class constructor, as in Listing 18.1.

LISTING 18.1. THE `CSpinner` CONSTRUCTOR.

```
 1: CSpinner::CSpinner()
 2: {
 3:     // Initialize the position, size, and color
 4:     m_iRadius = 0;
 5:     m_nMinute = 0;
 6:     m_crColor = 0;
 7:     // Nullify the pointers
 8:     m_pViewWnd = NULL;
 9:     m_bContinue = NULL;
10: }
```

For those variables that you need to be able to set and retrieve, your spinner class is simple enough that you can write all the set and get functions as inline functions in the class declaration. The color and position will be automatically calculated by the spinner object, so you don't need to add set functions for those two variables, but you do need to add set functions for the rest of the variables (not counting the color table). The only variables that you need to retrieve from the spinner object are the pointers to the view class and the check box variable. You can add all these functions to the `CSpinner` class declaration by opening the Spinner header file and adding the inline functions in Listing 18.2.

LISTING 18.2. THE `CSpinner` CLASS DECLARATION.

```
 1: class CSpinner : public CObject
 2: {
 3: public:
 4:     BOOL* GetContinue() {return m_bContinue;}
 5:     void SetContinue(BOOL* bContinue) { m_bContinue = bContinue;}
 6:     CWnd* GetViewWnd() { return m_pViewWnd;}
 7:     void SetViewWnd(CWnd* pWnd) { m_pViewWnd = pWnd;}
 8:     void SetLength(int iLength) { m_iRadius = iLength;}
 9:     void SetPoint(CPoint pPoint) { m_pCenter = pPoint;}
10:     CSpinner();
11:     virtual ~CSpinner();
12:
13: private:
14:     BOOL* m_bContinue;
15:     CWnd* m_pViewWnd;
16:     static COLORREF m_crColors[8];
17:     int m_iRadius;
18:     CPoint m_pCenter;
19:     int m_nMinute;
20:     int m_crColor;
21: };
```


Now that you have added all the support functions for setting and retrieving the necessary variables, you need to declare and populate the color table. This will look just like the color table definition you added to the drawing application on Day 10, “Creating Single Document Interface Applications.” The color table will consist of eight RGB values, with each value being either `0` or `255`, with every combination of these two settings. The best place to add this table declaration is in the spinner source code file, just before the class constructor, as in Listing 18.3.

LISTING 18.3. THE `CSpinner` COLOR TABLE.

```
 1: static char THIS_FILE[]=__FILE__;
 2: #define new DEBUG_NEW
 3: #endif
 4:
 5: COLORREF CSpinner::m_crColors[8] = {
 6:     RGB(   0,   0,   0),    // Black
 7:     RGB(   0,   0, 255),    // Blue
 8:     RGB(   0, 255,   0),    // Green
 9:     RGB(   0, 255, 255),    // Cyan
10:     RGB( 255,   0,   0),    // Red
11:     RGB( 255,   0, 255),    // Magenta
12:     RGB( 255, 255,   0),    // Yellow
13:     RGB( 255, 255, 255)        // White
```

continues

LISTING 18.3. CONTINUED

```
14: };
15:
16: ///////////////////////////////////////////////////////////////////////
17: // Construction/Destruction
18: ///////////////////////////////////////////////////////////////////////
19:
20: CSpinner::CSpinner()
21: {
22:     // Initialize the position, size, and color
23:     m_iRadius = 0;
24: .
25: .
26: .
```

Drawing the Spinner

Now comes the fun part: getting the spinner object to actually spin. To accomplish this, you'll calculate the new position of the starting and ending points of each line, set the view port origination point, select the drawing color, and create a pen to draw in that color. Once you have all of this, you will be able to draw the line from the starting point to the ending point. Once the line is drawn, you can restore the pen to what it was before the line was drawn. Next, you'll calculate the position of the next line to draw before exiting the function.

To add this functionality to your spinner object, add a member function to the `CSpinner` class. Specify the type as void, the name as `Draw`, and the access as public. Edit the function, adding the code in Listing 18.4.

LISTING 18.4. THE `CSpinner` `Draw` FUNCTION.

```
 1: void CSpinner::Draw()
 2: {
 3:     // Get a pointer to the device context
 4:     CDC *pDC = m_pViewWnd->GetDC();
 5:     // Set the mapping mode
 6:     pDC->SetMapMode (MM_LOENGLISH);
 7:     // Copy the spinner center
 8:     CPoint org = m_pCenter;
 9:     CPoint pStartPoint;
10:     // Set the starting point
11:     pStartPoint.x = (m_iRadius / 2);
12:     pStartPoint.y = (m_iRadius / 2);
13:     // Set the origination point
14:     org.x = m_pCenter.x + (m_iRadius / 2);
```

```
15:     org.y = m_pCenter.y + m_iRadius;
16:     // Set the viewport origination point
17:     pDC->SetViewportOrg(org.x, org.y);
18:
19:     CPoint pEndPoint;
20:     // Calculate the angle of the next line
21:     double nRadians = (double) (m_nMinute * 6) * 0.017453292;
22:     // Set the end point of the line
23:     pEndPoint.x = (int) (m_iRadius * sin(nRadians));
24:     pEndPoint.y = (int) (m_iRadius * cos(nRadians));
25:
26:
27:     // Create the pen to use
28:     CPen pen(PS_SOLID, 0, m_crColors[m_crColor]);
29:     // Select the pen for use
30:     CPen* pOldPen = pDC->SelectObject(&pen);
31:
32:     // Move to the starting point
33:     pDC->MoveTo (pEndPoint);
34:     // Draw the line to the end point
35:     pDC->LineTo (pStartPoint);
36:
37:     // Reselect the previous pen
38:     pDC->SelectObject(&pOldPen);
39:
40:     // Release the device context
41:     m_pViewWnd->ReleaseDC(pDC);
42:
43:     // Increment the minute
44:     if (++m_nMinute == 60)
45:     {
46:         // If the minutes have gone full circle, reset to 0
47:         m_nMinute = 0;
48:         // Increment the color
49:         if (++m_crColor == 8)
50:             // If we've gone through all colors, start again
51:             m_crColor = 0;
52:     }
53: }
```


That was quite a bit of code to type. What does it do? Well, to understand what this function is doing, and how it's going to make your spinner draw a color wheel on the window, let's take a closer look at the code.

To make efficient use of the spinner by the different threads, it'll only draw one line each time the function is called. This function will be called 60 times for each complete circle, once for each "minute" in the clockwise rotation. Each complete rotation will cause the spinner to switch to the next color in the color table.

One of the first things that you need to do in order to perform any drawing on the window is get the device context of the window. You do this by calling the `GetDC` function on the view object pointer:

```
CDC *pDC = m_pViewWnd->GetDC();
```

This function returns a `CDC` object pointer, which is an MFC class that encapsulates the device context.

Once you have a pointer to the device context, you can call its member function, `SetMapMode`, to set the mapping mode:

```
pDC->SetMapMode (MM_LOENGLISH);
```

The mapping mode determines how the x and y coordinates are translated into positions on the screen. The `MM_LOENGLISH` mode converts each logical unit to 0.01 inch on the screen. There are several different mapping modes, each converting logical units to different measurements on the screen.

At this point, you start preparing to draw the current line for the color wheel. You start by calculating the starting point for the line that will be drawn. This point will be consistent for all lines drawn by the spinner object. After you calculate the starting point for the line, you calculate the position of the viewport. The viewport is used as the starting point for the coordinates used for drawing.

The starting point for the line to be drawn is calculated in an off-center position. If you want the starting point for the lines to be in the center of the color wheel, set both the x and y coordinates of the starting point to 0.

Once the viewport origination point is calculated, use the `SetViewportOrg` function to set the viewport:

```
pDC->SetViewportOrg(org.x, org.y);
```

Now that you've got the drawing area specified, and the starting point for the line that you are going to be drawing, you need to figure out where the other end of the line will be. You'll perform this calculation using the following three lines of code:

```
double nRadians = (double) (m_nMinute * 6) * 0.017453292;
pEndPoint.x = (int) (m_iRadius * sin(nRadians));
pEndPoint.y = (int) (m_iRadius * cos(nRadians));
```

In the first of these calculations, convert the minutes into degrees, which can then be fed into the sine and cosine functions to set the x and y coordinates to draw a circle. This sets the end point of the line that will be drawn.

Now that you've figured out the starting and ending points of the line, you'll create a pen to use in drawing the line:

```
CPen pen(PS_SOLID, 0, m_crColors[m_crColor]);
```

You've specified that the pen will be solid and thin, and you are picking the current color from the color table. Once you create the pen to use, select the pen for drawing, being sure to capture the current pen as the return value from the device context object:

```
CPen* pOldPen = pDC->SelectObject(&pen);
```

Now you are ready to draw the line, which is done using the `MoveTo` and `LineTo` functions that you're well familiar with by now. Once the line is drawn, release the device context so that you don't have a resource leak in your application:

```
m_pViewWnd->ReleaseDC(pDC);
```

At this point, you've drawn the line, so all that's left to do is increment the minute counter, resetting it if you've made it all the way around the circle. Each time you complete a circle, you increment the color counter until you've gone through all eight colors, at which time you reset the color counter.

In order to be able to use the trigonometric functions in this function, include the `math.h` header file in the `Spinner` class source file. To add this, scroll up to the top of the source code file and add another `#include` line, specifying the `math.h` header file as the file to be included, as in Listing 18.5.

LISTING 18.5. THE `CSpinner` SOURCE FILE.

```
1: // Spinner.cpp : implementation of the CSpinner class
2: //
3: //////////////////////////////////////////////////////////////////////
4:
5: #include "stdafx.h"
6: #include <math.h>
7: #include "Tasking.h"
8: #include "Spinner.h"
```

Supporting the Spinners

Now that you've created the spinner class for drawing the spinning color wheel on the window, add some support for the spinners. You can add an array to hold the four

spinners in the document class, but you'll still need to calculate where each spinner should be placed on the application window and set all the variables in each of the spinners.

You can add all of this code to the document class, starting with the array of spinners. Add a member variable to the document class (in this instance, `CTaskingDoc`), specifying the type as `CSpinner`, the name as `m_cSpin[4]`, and the access as private. Once you add the array, open the source code to the document class and include the spinner header file, as in Listing 18.6.

LISTING 18.6. THE `CTaskingDoc` SOURCE FILE.

```
 1: // TaskingDoc.cpp : implementation of the CTaskingDoc class
 2: //
 3:
 4: #include "stdafx.h"
 5: #include "Tasking.h"
 6:
 7: #include "Spinner.h"
 8: #include "TaskingDoc.h"
 9: #include "TaskingView.h"
10: .
11: .
12: .
```

Calculating the Spinner Positions

One of the preparatory things that needs to happen while initializing the application is determining the locations of all four spinners. The window is roughly broken up into four quarters by the check boxes that will turn the spinner threads on and off, so it makes sense to divide the window area into four quarter squares and place one spinner in each quarter.

To calculate the location of each spinner, it is easiest to create a function that calculates the location for one spinner, placing the spinner into the quarter square appropriate for the spinner number. If the function was passed a pointer to the spinner object, it could update the spinner object directly with the location.

To add this functionality to your application, add a new member function to the document class (for instance, in the `CTaskingDoc` class). Specify the function type as void, the declaration as `CalcPoint(int nID, CSpinner *pSpin)`, and the access as private. Edit the function, adding the code in Listing 18.7.

LISTING 18.7. THE `CTaskingDoc CalcPoint` FUNCTION.

```
 1: void CTaskingDoc::CalcPoint(int nID, CSpinner *pSpin)
 2: {
 3:     RECT lWndRect;
 4:     CPoint pPos;
 5:     int iLength;
 6:     CTaskingView *pWnd;
 7:
 8:     // Get a pointer to the view window
 9:     pWnd = (CTaskingView*)pSpin->GetViewWnd();
10:     // Get the display area rectangle
11:     pWnd->GetClientRect(&lWndRect);
12:     // Calculate the size of the spinners
13:     iLength = lWndRect.right / 6;
14:     // Which spinner are we placing?
15:     switch (nID)
16:     {
17:     case 0:    // Position the first spinner
18:         pPos.x = (lWndRect.right / 4) - iLength;
19:         pPos.y = (lWndRect.bottom / 4) - iLength;
20:         break;
21:     case 1:    // Position the second spinner
22:         pPos.x = ((lWndRect.right / 4) * 3) - iLength;
23:         pPos.y = (lWndRect.bottom / 4) - iLength;
24:         break;
25:     case 2:    // Position the third spinner
26:         pPos.x = (lWndRect.right / 4) - iLength;
27:         pPos.y = ((lWndRect.bottom / 4) * 3) - (iLength * 1.25);
28:         break;
29:     case 3:    // Position the fourth spinner
30:         pPos.x = ((lWndRect.right / 4) * 3) - iLength;
31:         pPos.y = ((lWndRect.bottom / 4) * 3) - (iLength * 1.25);
32:         break;
33:     }
34:     // Set the size of the spinner
35:     pSpin->SetLength(iLength);
36:     // Set the location of the spinner
37:     pSpin->SetPoint(pPos);
38: }
```

In this function, the first thing that you do is move the pointer to the view window from the spinner object by calling the `GetViewWnd` function:

```
pWnd = (CTaskingView*)pSpin->GetViewWnd();
```

By moving the pointer directly from the spinner object, you save a few steps by taking a more direct route to get the information that you need.

Once you have a pointer to the view object, you can call the window's `GetClientRect` function to get the size of the available drawing area:

```
pWnd->GetClientRect(&lWndRect);
```

Once you have the size of the drawing area, you can calculate a reasonable color wheel size by dividing the length of the drawing area by 6:

```
iLength = lWndRect.right / 6;
```

Dividing the drawing area by 4 will position you at the middle of the upper-left square. Subtract the size of the circle from this point, and you have the upper-left corner of the drawing area for the first spinner:

```
pPos.x = (lWndRect.right / 4) - iLength;
pPos.y = (lWndRect.bottom / 4) - iLength;
```

You can then include variations on this position, mostly by multiplying the center of the quadrant by 3 to move it to the center of the right or lower quadrant, and you can calculate the positions of the other three spinners.

Once you calculate the length and position for the spinner, you call the `SetLength` and `SetPoint` functions to pass these values to the spinner that they have been calculated for:

```
pSpin->SetLength(iLength);
pSpin->SetPoint(pPos);
```

Initializing the Spinners

Because you wrote the previous function to calculate the location of each spinner on the window to work on only one spinner each time it is called, you need some routine that will initialize each spinner, calling the previous function once for each spinner. You need this function to get a pointer to the view object and pass that along to the spinner. You also need to get pointers to the check box variables for the spinners that will be used by the independently running threads. Your code can do all this by just looping through the array of spinners, setting both of these pointers for each spinner, and then passing the spinner to the function you just finished.

To create this function for your application, add a new member function to the document class (`CTaskingDoc` in this instance). Specify the type as void, and give the function a suitable name (for instance, `InitSpinners`), and then specify the access as private because you'll only need to call this function once when the application is starting. Edit the new function, adding the code in Listing 18.8.

LISTING 18.8. THE `CTaskingDoc InitSpinners` FUNCTION.

```
 1: void CTaskingDoc::InitSpinners()
 2: {
 3:     int i;
 4:
 5:     // Get the position of the view
 6:     POSITION pos = GetFirstViewPosition();
 7:     // Did we get a valid position?
 8:     if (pos != NULL)
 9:     {
10:         // Get a pointer to the view
11:         CView* pView = GetNextView(pos);
12:
13:         // Loop through the spinners
14:         for (i = 0; i < 4; i++)
15:         {
16:             // Set the pointer to the view
17:             m_cSpin[i].SetViewWnd(pView);
18:             // Initialize the pointer to the continuation indicator
19:             m_cSpin[i].SetContinue(NULL);
20:             switch (i)
21:             {
22:             case 1:    // Set the pointer to the first thread
23:                        // continuation indicator
24:                 m_cSpin[i].SetContinue(&((CTaskingView*)pView)-
   ➥>m_bThread1);
25:                 break;
26:             case 3:    // Set the pointer to the second thread
27:                        // continuation indicator
28:                 m_cSpin[i].SetContinue(&((CTaskingView*)pView)-
   ➥>m_bThread2);
29:                 break;
30:             }
31:             // Calculate the location of the spinner
32:             CalcPoint(i, &m_cSpin[i]);
33:         }
34:     }
35: }
```

In this function, you first went through the steps of getting a pointer to the view class from the document, as you did initially back on Day 10. Once you have a valid pointer to the view, start a loop to initialize each of the spinners in the array. You call the `SetViewWnd` spinner function to set the spinner's pointer to the view window and then initialize the spinner's pointer to the check box variable to `NULL` for all spinners. If the spinner is either of the two that will be used by independent threads, you pass a pointer to the appropriate check box variable. Once you set all of this, call the `CalcPoint`

function that you created just a few minutes earlier to calculate the location of the spinner on the view window.

Although you've seen several examples of using pointers, the way that you are passing a pointer to the check box variable to the spinner deserves taking a closer look:

```
m_cSpin[i].SetContinue(&((CTaskingView*)pView)->m_bThread1);
```

In this statement, you take the pointer to the view object, `pView`, which is a pointer for a CView object, and cast it as a pointer to the specific view class that you have created in your application:

```
(CTaskingView*)pView
```

Now that you can treat the pointer to the view object as a `CTaskingView` object, you can get to the check box variable, `m_bThread1`, which is a public member of the `CTaskingView` class:

```
((CTaskingView*)pView)->m_bThread1
```

Once you access the `m_bThread1` variable, get the address of this variable by placing an ampersand in front of this whole string:

```
&((CTaskingView*)pView)->m_bThread1
```

Passing this address for the `m_bThread1` variable to the `SetContinue` function, you are, in effect, passing a pointer to the `m_bThread1` variable, which can be used to set the pointer to this variable that the spinner object contains.

Now that you've created the routines to initialize all the spinners, make sure that this routine is called when the application is started. The best place to put this logic is the `OnNewDocument` function in the document class. This function will be called when the application is started, so it is a logical place to trigger the initialization of the spinner objects. To add this code to the `OnNewDocument` function, add the code in Listing 18.9 to the `OnNewDocument` function in the document class.

LISTING 18.9. THE `CTaskingDoc OnNewDocument` FUNCTION.

```
1: BOOL CTaskingDoc::OnNewDocument()
2: {
3:     if (!CDocument::OnNewDocument())
4:         return FALSE;
5:
6:     // TODO: add reinitialization code here
7:     // (SDI documents will reuse this document)
```

```
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Initialize the spinners
14:     InitSpinners();
15:
16:     ///////////////////////
17:     // MY CODE ENDS HERE
18:     ///////////////////////
19:
20:     return TRUE;
21: }
```

Spinning the Spinner

Once last thing that you'll add to the document class for now is a means of calling the `Draw` function for a specific spinner from outside the document class. Because the array of spinners was declared as a private variable, no outside objects can get access to the spinners, so you need to add access for the outside objects. You can add a function to provide this access by adding a new member function to your document class. Specify the function type as void, specify the function declaration with a name and a single integer argument for the spinner number, such as `DoSpin(int nIndex)`, and then specify the function's access as public. Once you have added the function, you can add the code in Listing 18.10 to the function to perform the actual call to the specified spinner.

LISTING 18.10. THE `CTaskingDoc DoSpin` FUNCTION.

```
1: void CTaskingDoc::DoSpin(int nIndex)
2: {
3:     // Spin the Spinner
4:     m_cSpin[nIndex].Draw();
5: }
```

Adding the `OnIdle` Tasks

Now that you have the supporting functionality in place, it's time to turn your attention to adding the various threads that will turn the various spinners. The first threads to add are the ones executing while the application is idle. You'll add a clicked event handler for the two On Idle check boxes so that you can keep the variables for these two check boxes in sync with the window. You'll also add the code to the application's `OnIdle` function to run these two spinners when the application is idle and the check boxes for these two spinner threads are checked.

The use of the term *thread* in the preceding is slightly misleading. Any functionality that you place in the `OnIdle` function is running in the main application thread. All the `OnIdle` processing that you add to the sample application won't be running as an independent thread, but will be just functions that can be called from the main thread.

Starting and Stopping the `OnIdle` Tasks

The `OnIdle` function will check the values of the two check box variables that specify whether each should run, so all your application needs to do when either of these check boxes is clicked is make sure that the variables in the view object are synchronized with the controls on the window. All that you need to do to accomplish this is call the `UpdateData` function when either of these controls is clicked. You need to be able to start and stop the `OnIdle` tasks by adding a single event handler for both of the On Idle Thread check boxes and then calling the `UpdateData` function in this event function.

To add this to your application, open the Class Wizard and select the view class (in this case, `CTaskingView`). Select one of the On Idle check boxes and add a function for the `BN_CLICKED` event. Change the name of the suggested function to `OnCbonidle` and click OK. Do the same thing for the other On Idle check box. Once you specify that both of these events use the same code, click on the Edit Code button and add the code in Listing 18.11.

LISTING 18.11. THE `CTaskingView` `OnCbonidle` FUNCTION.

```
 1: void CTaskingView::OnCbonidle()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Sync the variables with the dialog
10:     UpdateData(TRUE);
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
```

Building the `OnIdle` Threads

If you examine the application class (`CTaskingApp`) source code, you'll find that the `OnIdle` function isn't there. All the functionality that the OnIdle function needs to perform by default is in the ancestor class of the application class that was created for your project. The only reason to have an `OnIdle` function in your application class is that your application needs some specific functionality to be performed during this event. As a result, you need to specifically add this event handler to your application using the Class Wizard.

Once you add the `OnIdle` function to your application class, what does it need to do? First, it needs to get a pointer to the view so that it can check the status of the check box variables. Next, it needs to get a pointer to the document class so that it can call the `DoSpin` function to trigger the appropriate spinner object. The key to both of these actions is getting pointers to each of these objects. When you begin looking at what is necessary to get these pointers, you'll find that you have to reverse the order in which you get the pointers. You need to get a pointer to the document object in order to get a pointer to the view. However, to get a pointer to the document, you have to go through the document template, getting a pointer to the template before you can get a pointer to the document. Each of these steps requires the same sequence of events, first getting the position of the first object and then getting a pointer to the object in that position. What you'll do is get the position of the first document template and then get a pointer to the document template in that position. Next, you'll use the document template to get the position of the first document and then use the document template to get a pointer to the document in that first position. Finally, you'll use the document to get the position of the first view and then use the document again to get a pointer to the view in the position specified. Once you've got a pointer to the view, you can check the value of the check boxes and call the appropriate spinner.

18

To add this functionality to your application, use the Class Wizard to add a function to the `OnIdle` event message for the application class (in this case, `CTaskingApp`). Once you add the function, click the Edit Code button and add the code in Listing 18.12.

LISTING 18.12. THE `CTaskingApp` `OnIdle` FUNCTION.

```
1: BOOL CTaskingApp::OnIdle(LONG lCount)
2: {
3:     // TODO: Add your specialized code here and/or call the base class
4:
5:     // Get the position of the first document template
6:     POSITION pos = GetFirstDocTemplatePosition();
7:     // Do we have a valid template position?
8:     if (pos)
```

continues

LISTING 18.12. CONTINUED

```
 9:     {
10:         // Get a pointer to the document template
11:         CDocTemplate* pDocTemp = GetNextDocTemplate(pos);
12:         // Do we have a valid pointer?
13:         if (pDocTemp)
14:         {
15:             // Get the position of the first document
16:             POSITION dPos = pDocTemp->GetFirstDocPosition();
17:             // Do we have a valid document position?
18:             if (dPos)
19:             {
20:                 // Get a pointer to the document
21:                 CTaskingDoc* pDocWnd =
22:                     (CTaskingDoc*)pDocTemp->GetNextDoc(dPos);
23:                 // Do we have a valid pointer?
24:                 if (pDocWnd)
25:                 {
26:                     // Get the position of the view
27:                     POSITION vPos = pDocWnd->GetFirstViewPosition();
28:                     // Do we have a valid view position?
29:                     if (vPos)
30:                     {
31:                         // Get a pointer to the view
32:                         CTaskingView* pView =
        ➥(CTaskingView*)pDocWnd->GetNextView(vPos);
33:                         // Do we have a valid pointer?
34:                         if (pView)
35:                         {
36:                             // Should we spin the first idle thread?
37:                             if (pView->m_bOnIdle1)
38:                                 // Spin the first idle thread
39:                                 pDocWnd->DoSpin(0);
40:                             // Should we spin the second idle thread?
41:                             if (pView->m_bOnIdle2)
42:                                 // Spin the second idle thread
43:                                 pDocWnd->DoSpin(2);
44:                         }
45:                     }
46:                 }
47:             }
48:         }
49:     }
50:
51:     // Call the ancestor's idle processing
52:     return CWinApp::OnIdle(lCount);
53: }
```

If you compile and run your application now, you should be able to check either of the On Idle Thread check boxes, and see the spinner drawing a color wheel, as shown in

Figure 18.7, as long as you are moving the mouse. However, the moment you let the application become totally idle—no mouse movement or anything else—the spinner will stop spinning.

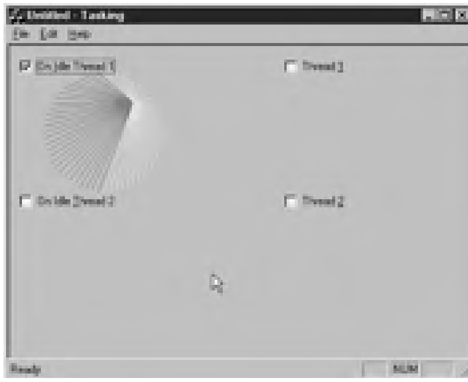


FIGURE 18.7.
On Idle Thread drawing a color wheel.

Making the `OnIdle` Tasks Continuous

It's not very practical to keep moving the mouse around to make your application continue performing the tasks that it's supposed to do when the application is idle. There must be a way to get the application to continue to call the `OnIdle` function as long as the application is idle. Well, there is. If you look at the last line in the `OnIdle` function, you'll notice that the `OnIdle` function returns the result value from the ancestor `OnIdle` function. It just so happens that this function returns `FALSE` as soon as there is no `OnIdle` functionality to be performed.

You want the `OnIdle` function to always return `TRUE`. This will cause the `OnIdle` function to continue to be called, over and over, whenever the application is idle. If you move the call to the ancestor `OnIdle` function to the first part of the function and then return `TRUE`, as in Listing 18.13, you will get your spinner to continue turning, no matter how long the application sits idle.

LISTING 18.13. THE MODIFIED `CTaskingApp OnIdle` FUNCTION.

```
1: BOOL CTaskingApp::OnIdle(LONG lCount)
2: {
3:     // TODO: Add your specialized code here and/or call the base class
4:
5:     // Call the ancestor's idle processing
6:     CWinApp::OnIdle(lCount);
7:
```

continues

LISTING 18.13. CONTINUED

```
 8:     // Get the position of the first document template
 9:     POSITION pos = GetFirstDocTemplatePosition();
10:     // Do we have a valid template position?
11:     if (pos)
12:     {
.
.
.
51:     }
52:     return TRUE;
53: }
```

If you compile and run your application, you can turn on the `OnIdle` tasks and see them continue to turn, even when you are not moving the mouse. However, if you activate any of the menus, or if you open the About window, both of these tasks come to a complete stop, as in Figure 18.8. The reason is that the open menus, and any open modal dialog windows, prevent the `OnIdle` function from being called. One of the limitations of `OnIdle` processing is that certain application functionality prevents it from being performed.

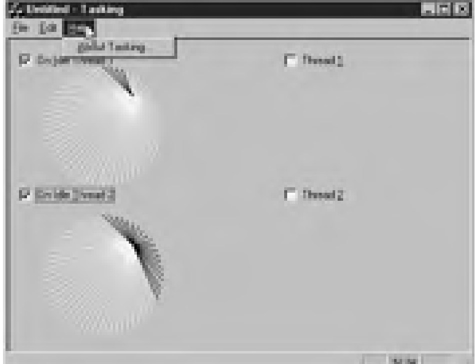


FIGURE 18.8.
On Idle Thread stopped by the menu.

Adding Independent Threads

Now that you've seen what is involved in adding an `OnIdle` task, it's time to see what's involved in adding an independent thread to your application. To add a thread to your application, you'll add a main function for the threads. You'll also add the code to start and stop the threads. Finally, you'll add the code to the independent thread check boxes to start and stop each of these threads.

Creating the Main Thread Function

Before you can spin off any independent threads, the thread must know what to do. You will create a main thread function to be executed by the thread when it starts. This function will act as the main function for the thread, and the thread will end once the function ends. Therefore, this function must act as the primary control of the thread, keeping the thread running as long as there is work for the thread to do and then exiting once the thread's work is completed.

When you create a function to be used as the main function for a thread, you can pass a single parameter to this function. This parameter is a pointer to anything that contains all the information the thread needs to perform its tasks. For the application you've been building in this chapter, the parameter can be a pointer to the spinner that the thread will operate. Everything else that the thread needs can be extracted from the spinner object.

Once the thread has a pointer to its spinner, it can get a pointer to the check box variable that tells it whether to continue spinning or stop itself. As long as the variable is `TRUE`, the thread should continue spinning.

To add this function to your application, add a new member function to the document class in your application. Specify the function type as `UINT`, the function declaration as `ThreadFunc(LPVOID pParam)`, and the access as private. You'll start the thread from within the document class, so there's no need for any other classes to see this function. Once you've added this function, edit it with the code in Listing 18.14.

LISTING 18.14. THE `CTaskingDoc` `ThreadFunc` FUNCTION.

```
 1: UINT CTaskingDoc::ThreadFunc(LPVOID pParam)
 2: {
 3:     // Convert the argument to a pointer to the
 4:     // spinner for this thread
 5:     CSpinner* lpSpin = (CSpinner*)pParam;
 6:     // Get a pointer to the continuation flag
 7:     BOOL* pbContinue = lpSpin->GetContinue();
 8:
 9:     // Loop while the continue flag is true
10:     while (*pbContinue)
11:         // Spin the spinner
12:         lpSpin->Draw();
13:     return 0;
14: }
```

Starting and Stopping the Threads

Now that you have a function to call for the independent threads, you need some way of controlling the threads, starting and stopping them. You need to be able to hold onto a couple of pointers for `CWinThread` objects, which will encapsulate the threads. You'll add these pointers as variables to the document object and then use them to capture the return variable from the `AfxBeginThread` function that you will use to start both of the threads.

To add these variables to your application, add a new member variable to your document class. Specify the variable type as `CWinThread*`, the variable name as `m_pSpinThread[2]`, and the variable access as private. This will provide you with a two slot array for holding these variables.

Now that you have a place to hold the pointers to each of the two threads, you'll add the functionality to start the threads. You can add a single function to start either thread, if it's not currently running, or to wait for the thread to stop itself, if it is running. This function will need to know which thread to act on and whether to start or stop the thread.

To add this functionality, add a new member function to the document class. Specify the function type as void, the function declaration as `SuspendSpinner(int nIndex, BOOL bSuspend)`, and the function access as public, and check the Static check box. Edit the code for this function, adding the code in Listing 18.15.

LISTING 18.15. THE `CTaskingDoc SuspendSpinner` FUNCTION.

```
 1: void CTaskingDoc::SuspendSpinner(int nIndex, BOOL bSuspend)
 2: {
 3:     // if suspending the thread
 4:     if (!bSuspend)
 5:     {
 6:         // Is the pointer for the thread valid?
 7:         if (m_pSpinThread[nIndex])
 8:         {
 9:             // Get the handle for the thread
10:             HANDLE hThread = m_pSpinThread[nIndex]->m_hThread;
11:             // Wait for the thread to die
12:             ::WaitForSingleObject (hThread, INFINITE);
13:         }
14:     }
15:     else    // We are running the thread
16:     {
17:         int iSpnr;
18:         // Which spinner to use?
19:         switch (nIndex)
20:         {
21:         case 0:
```

```
22:             iSpnr = 1;
23:             break;
24:         case 1:
25:             iSpnr = 3;
26:             break;
27:         }
28:         // Start the thread, passing a pointer to the spinner
29:         m_pSpinThread[nIndex] = AfxBeginThread(ThreadFunc,
30:             (LPVOID)&m_cSpin[iSpnr]);
31:     }
32: }
```

The first thing that you do in this function is check to see if the thread is being stopped or started. If the thread is being stopped, check to see if the pointer to the thread is valid. If the pointer is valid, you retrieve the handle for the thread by reading the value of the handle property of the `CWinThread` class:

```
HANDLE hThread = m_pSpinThread[nIndex]->m_hThread;
```

Once you have the handle, you use the handle to wait for the thread to stop itself with the `WaitForSingleObject` function.

```
::WaitForSingleObject (hThread, INFINITE);
```


The `WaitForSingleObject` function is a Windows API function that tells the operating system you want to wait until the thread, whose handle you are passing, has stopped. The second argument to this function specifies how long you are willing to wait. By specifying `INFINITE`, you tell the operating system that you will wait forever, until this thread stops. If you specify a timeout value, and the thread does not stop by the time you specify, the function returns a value that indicates whether the thread has stopped. Because you specify `INFINITE` for the timeout period, you don't need to worry about capturing the return value because this function does not return until the thread stops.

If the thread is being started, you determine which spinner to use and then start that thread by calling the `AfxBeginThread` function.

```
m_pSpinThread[nIndex] = AfxBeginThread(ThreadFunc,
            (LPVOID)&m_cSpin[iSpnr]);
```

You passed the function to be called as the main function for the thread and the address of the spinner to be used by that thread.

Triggering the Threads from the View Object

Now that you have a means of starting and stopping each of the independent threads, you need to be able to trigger the starting and stopping from the check boxes on the window.

When each of the two check boxes is checked, you'll start each of the threads. When the check boxes are unchecked, each of the threads must be stopped. The second part of this is easy: As long as the variable tied to the check box is kept in sync with the control, once the check box is unchecked, the thread will stop itself. However, when the check box is checked, you'll need to call the document function that you just created to start the thread.

To add this functionality to the first of the two thread check boxes, use the Class Wizard to add a function to the `BN_CLICKED` event for the check box. Once you add the function, edit it with the code in Listing 18.16.

LISTING 18.16. THE `CTaskingView` `OnCbthread1` FUNCTION.

```
 1: void CTaskingView::OnCbthread1()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Sync the variables with the dialog
10:     UpdateData(TRUE);
11:
12:     // Get a pointer to the document
13:     CTaskingDoc* pDocWnd = (CTaskingDoc*)GetDocument();
14:     // Did we get a valid pointer?
15:     ASSERT_VALID(pDocWnd);
16:
17:     // Suspend or start the spinner thread
18:     pDocWnd->SuspendSpinner(0, m_bThread1);
19:
20:     ///////////////////////
21:     // MY CODE ENDS HERE
22:     ///////////////////////
23: }
```

In this function, the first thing that you do is to call `UpdateData` to keep the variables in sync with the controls on the window. Next, you retrieve a pointer to the document. Once you have a valid pointer, you call the document's `SuspendSpinner` function, specifying the first thread and passing the current value of the variable tied to this check box to indicate whether the thread is to be started or stopped.

To add this same functionality to the other thread check box, perform the same steps, adding the code in Listing 18.17.

LISTING 18.17. THE `CTaskingView` `OnCbthread2` FUNCTION.

```
 1: void CTaskingView::OnCbthread2()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Sync the variables with the dialog
10:     UpdateData(TRUE);
11:
12:     // Get a pointer to the document
13:     CTaskingDoc* pDocWnd = (CTaskingDoc*)GetDocument();
14:     // Did we get a valid pointer?
15:     ASSERT_VALID(pDocWnd);
16:
17:     // Suspend or start the spinner thread
18:     pDocWnd->SuspendSpinner(1, m_bThread2);
19:
20:     ///////////////////////
21:     // MY CODE ENDS HERE
22:     ///////////////////////
23: }
```


Now that you've added the ability to start and stop the independent threads, compile and run your application. You'll see that you can start and stop the independent threads with their check boxes, as well as the `OnIdle` tasks.

At this point, if you play around with your application for a while, you'll notice a bit of a difference between the two types of threads. If you have all threads running and are actively moving the mouse, you might notice the `OnIdle` spinners slowing down in their spinning (unless you have a very fast machine). The independent threads are taking a good deal of the processor time away from the main application thread, leaving less processor time to be idle. As a result, it's easier to keep your application busy. The other thing that you might notice is that if you activate the menus or open the About window, although the `OnIdle` tasks come to a complete stop, the independent threads continue to run, as in Figure 18.9. These two threads are completely independent processes running within your application, so they are not affected by the rest of the application.

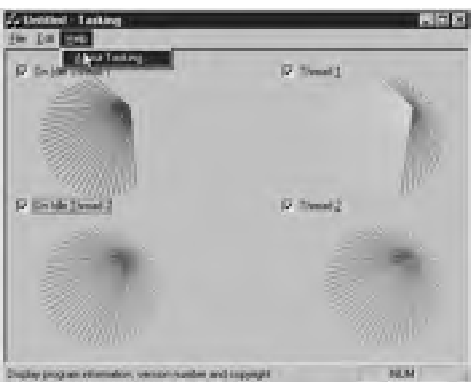


FIGURE 18.9.
The threads are not affected by the menu.

Shutting Down Cleanly

You might think that you are done with this application until you try to close the application while one or both of the independent threads is running. You'll see an unpleasant notification that you still have some work to do, as in Figure 18.10. It seems that leaving the threads running when you closed the application caused it to crash.

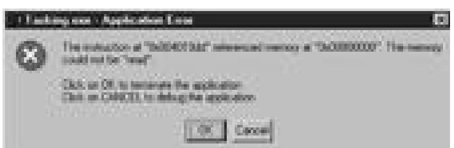


FIGURE 18.10.
Application error notification.

Even though the application was closing, the threads were continuing to run. When these threads checked the value of the variable indicating whether to continue running or spin their spinners, they were trying to access a memory object that no longer existed. This problem causes one of the most basic and most fatal application memory errors, which you should eliminate before allowing anyone else to use the application.

What you need to do to prevent this error is stop both of the threads before allowing the application to close. The logical place to take this action is the `OnDestroy` event message processing in the view class. This event message is sent to the view class to tell it to clean up anything that it needs to clean up before closing the application. You can add code to set both of the check box variables to `FALSE` so that the threads will stop themselves and then call the `SuspendSpinner` function for each thread to make sure that both threads have stopped before allowing the application to close. You do not need to call `UpdateData` to sync the variables with the controls because the user doesn't need to see when you've change the value of either check box.

To add this functionality to your application, add an event-handler function for the `OnDestroy` event message to the view class. This function does not normally exist in the view class that is created by the AppWizard, so you need to add it when it is needed in the descendent view class. Edit the function, adding the code in Listing 18.18.

LISTING 18.18. THE `CTaskingView` `OnDestroy` FUNCTION.

```
 1: void CTaskingView::OnDestroy()
 2: {
 3:     CFormView::OnDestroy();
 4:
 5:     // TODO: Add your message handler code here
 6:
 7:     ///////////////////////
 8:     // MY CODE STARTS HERE
 9:     ///////////////////////
10:
11:     // Is the first thread running?
12:     if (m_bThread1)
13:     {
14:         // Specify to stop the first thread
15:         m_bThread1 = FALSE;
16:         // Get a pointer to the document
17:         CTaskingDoc* pDocWnd = (CTaskingDoc*)GetDocument();
18:         // Did we get a valid pointer?
19:         ASSERT_VALID(pDocWnd);
20:
21:         // Suspend the spinner thread
22:         pDocWnd->SuspendSpinner(0, m_bThread1);
23:     }
24:     // Is the second thread running?
25:     if (m_bThread2)
26:     {
27:         // Specify to stop the second thread
28:         m_bThread2 = FALSE;
29:         // Get a pointer to the document
30:         CTaskingDoc* pDocWnd = (CTaskingDoc*)GetDocument();
31:         // Did we get a valid pointer?
32:         ASSERT_VALID(pDocWnd);
33:
34:         // Suspend the spinner thread
35:         pDocWnd->SuspendSpinner(1, m_bThread2);
36:     }
37:
38:     ///////////////////////
39:     // MY CODE ENDS HERE
40:     ///////////////////////
41: }
```

18

In this function, you do exactly what you need to do. You check first one check box variable and then the other. If either is `TRUE`, you set the variable to `FALSE`, get a pointer to the document, and call the `SuspendSpinner` function for that thread. Now when you close your application while the independent threads are running, your application will close without crashing.

Summary

Today, you learned quite a bit. You learned about the different ways you can make your applications perform multiple tasks at one time. You also learned about some of the considerations to take into account when adding this capability to your applications. You learned how to make your application perform tasks when the application is sitting idle, along with some of the limitations and drawbacks associated with this approach. You also learned how to create independent threads in your application that will perform their tasks completely independently of the rest of the application. You implemented an application that uses both of these approaches so that you could experience how each approach works.

> **Tip:** When you start adding multitasking capabilities to your applications to perform separate tasks, be aware that this is a very advanced aspect of programming for Windows. You need to understand a lot of factors and take into account far more than we can reasonably cover in a single day. If you want to build applications using this capability, get an advanced book on programming Windows applications with MFC or Visual C++. The book should include a substantial section devoted to multithreading with MFC and cover all the synchronization classes in much more detail than we did here. Remember that you need a book that focuses on MFC, not the Visual C++ development environment. (MFC is supported by most commercial C++ development tools for building Windows applications, including Borland and Symantec's C++ compilers, so coverage for this topic extends beyond the Visual C++ environment.)

Q&A

Q How can I use the other version of the `AfxBeginThread` to encapsulate a thread in a custom class?

A First, the other version of `AfxBeginThread` is primarily for creating user-interface threads. The version that you used in today's sample application is for creating what are called worker threads that immediately take off on a specific task. If you

want to create a user-interface thread, you need to inherit your custom class from the `CWinThread` class. Next, override several ancestor functions in your custom class. Once the class is ready to use, you use the `RUNTIME_CLASS` macro to get a pointer to the runtime class of your class and pass this pointer to the `AfxBeginThread` function, as follows:

```
CWinThread* pMyThread =
            AfxBeginThread(RUNTIME_CLASS(CMyThreadClass));
```

Q Can I use `SuspendThread` and `ResumeThread` to start and stop my independent threads in my sample application?

A Yes, but you need to make a few key changes to your application. First, in the `OnNewDocument` function, initialize the two thread pointers to `NULL`, as shown in Listing 18.19.

LISTING 18.19. THE MODIFIED `CTaskingDoc` `OnNewDocument` FUNCTION.

```
 1: BOOL CTaskingDoc::OnNewDocument()
 2: {
 3:     if (!CDocument::OnNewDocument())
 4:         return FALSE;
 5:
 6:     // TODO: add reinitialization code here
 7:     // (SDI documents will reuse this document)
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Initialize the spinners
14:     InitSpinners();
15:
16:     // Initialize the thread pointers
17:     m_pSpinThread[0] = NULL;
18:     m_pSpinThread[1] = NULL;
19:
20:     ///////////////////////
21:     // MY CODE ENDS HERE
22:     ///////////////////////
23:
24:     return TRUE;
25: }
```

Next, modify the thread function so that the thread does not stop itself when the check box variable is `FALSE` but continues to loop, as shown in Listing 18.20.

LISTING 18.20. THE MODIFIED CTaskingDoc ThreadFunc FUNCTION.

```
 1: UINT CTaskingDoc::ThreadFunc(LPVOID pParam)
 2: {
 3:     // Convert the argument to a pointer to the
 4:     // spinner for this thread
 5:     CSpinner* lpSpin = (CSpinner*)pParam;
 6:     // Get a pointer to the continuation flag
 7:     BOOL* pbContinue = lpSpin->GetContinue();
 8:
 9:     // Loop while the continue flag is true
10:     while (TRUE)
11:         // Spin the spinner
12:         lpSpin->Draw();
13:     return 0;
14: }
```

Finally, modify the `SuspendSpinner` function so that if the thread pointer is valid, it calls the `SuspendThread` function on the thread pointer to stop the thread and the `ResumeThread` function to restart the thread, as shown in Listing 18.21.

LISTING 18.21. THE MODIFIED CTaskingDoc SuspendSpinner FUNCTION.

```
 1: void CTaskingDoc::SuspendSpinner(int nIndex, BOOL bSuspend)
 2: {
 3:     // if suspending the thread
 4:     if (!bSuspend)
 5:     {
 6:         // Is the pointer for the thread valid?
 7:         if (m_pSpinThread[nIndex])
 8:         {
 9:             // Suspend the thread
10:             m_pSpinThread[nIndex]->SuspendThread();
11:         }
12:     }
13:     else    // We are running the thread
14:     {
15:         // Is the pointer for the thread valid?
16:         if (m_pSpinThread[nIndex])
17:         {
18:             // Resume the thread
19:             m_pSpinThread[nIndex]->ResumeThread();
20:         }
21:         else
22:         {
23:             int iSpnr;
24:             // Which spinner to use?
25:             switch (nIndex)
```

```
26:             {
27:             case 0:
28:                 iSpnr = 1;
29:                 break;
30:             case 1:
31:                 iSpnr = 3;
32:                 break;
33:             }
34:             // Start the thread, passing a pointer to the spinner
35:             m_pSpinThread[nIndex] = AfxBeginThread(ThreadFunc,
36:                 (LPVOID)&m_cSpin[iSpnr]);
37:         }
38:     }
39: }
```

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. When is the `OnIdle` function called?
2. How can you cause the `OnIdle` function to be repeatedly called while the application is sitting idle?
3. What is the difference between an `OnIdle` task and a thread?
4. What are the four thread synchronization objects?
5. Why shouldn't you specify a higher than normal priority for the threads in your application?

Exercises

1. If you open a performance monitor on your system while the application that you built today is running, you'll find that even without any of the threads running, the processor usage remains 100 percent, as shown in Figure 18.11. The `OnIdle` function is continuously being called even when there is nothing to be done.

   Modify the `OnIdle` function so that if there's nothing to be done, neither of the `OnIdle` tasks are active. Then, the `OnIdle` function will not continue to be called until one of these threads is active, at which time it should be continuously called until both threads are once again turned off. This will allow the processor to drop to a minimal utilization, as shown in Figure 18.12.

FIGURE 18.11.
Processor utilization at 100 percent.

FIGURE 18.12.
Processor utilization at normal levels.

2. When starting the independent threads, give one of the threads a priority of `THREAD_PRIORITY_NORMAL` and the other a priority of `THREAD_PRIORITY_LOWEST`.

DAY 19

Building Your Own Widgets—Creating ActiveX Controls

The software industry has seen a revolution over the past couple years. How software is built and packaged has moved from a model where all applications are large, monolithic pieces of executable code to a model where most applications consist of small building blocks. These small building blocks, often called components, can be created using any of several different languages and can take many different forms. One of the most popular components is the ActiveX control. If you know how to create your own ActiveX controls, you can build your own components and provide them to other programmers. Today, you will learn

- How to use the Visual C++ wizards to build ActiveX controls.
- How to add properties and methods to your controls using the Class Wizard.
- How to test your control using the tools provided with Visual C++.

What Is an ActiveX Control?

An ActiveX control is a set of functionality packaged in a COM (Component Object Model) object. This COM object is self-contained, although it does not have the ability to run by itself. An ActiveX control can only run within a ActiveX container, such as a Visual C++ or Visual Basic application.

As you learned on Day 9, "Adding ActiveX Controls to Your Application," ActiveX controls provide a series of interfaces used by the container application to trigger the various sets of functionality contained in the control. Many of these interfaces are used for triggering events in the control or in the containing application. Others are for specifying the property page of the control or for communicating whether the control has been activated. All in all, so many interfaces are built into most ActiveX controls that coding the functionality for each of these interfaces yourself would take quite some time. Luckily, the Visual C++ App and Class Wizards add much of this functionality for you, allowing you to focus on the specific functionality that the control is supposed to have.

Among the aspects of the control you create that you still must plan yourself are what properties, methods, and events you will expose for your control. You can add these elements to your control through the Class Wizard, but if any of the properties or events require special code on your part, then you must add it yourself. As should be expected with any methods that you add to your controls, you have to supply all of the code. The Class Wizard will add the surrounding structure and code to allow the containing application to see and call the method, just as it will add all the code necessary to call any event handlers for your applications.

Properties

Properties are attributes of controls that are visible to, and often modifiable by, the container application. The four basic types of properties are ambient, extended, stock, and custom. Ambient properties are provided by the container application to the control—such things as background color or the default font to be used—so that the control looks like part of the container application. Extended properties are not actually properties of the control but instead are provided and implemented by the container application, such as tab order. The control may extend these properties somewhat; for example, if the control contains two or more standard controls, it may control the tab order within the overall control, returning the tab order control to the application once the control has completed its internal tab order. Stock properties are implemented by the ActiveX control development kit, such as control font or control background color. The final type of properties, custom properties, are what you are most concerned with because these

properties are specific to your control and are directly related to the functionality of your control.

You can specify any properties you need in your control using the Automation tab on the Class Wizard. When you add a new property to your control through the Class Wizard, you'll specify several aspects of the property.

The first aspect is the external property name, which is the name shown to the containing application for the property. Another aspect that you can specify is the internal variable name, which is used in your code, but only if the property is implemented as a member variable. You also specify the variable type for the property.

If you specify that the property is to be implemented as a member variable (the property is a member variable of the control class), then you can specify the name of the notification function, which is called when the property is changed by the containing application. If the property is not a member variable of the control class, you need to specify that it is altered and viewed through `Get` and `Set` methods, where the containing application calls a `Get` method to get the current value of the property and calls a `Set` method to change the value of the property. If the property is maintained through `Get` and `Set` methods, then you can specify the names of these two methods.

For all these aspects of a property, the Add Property dialog suggests appropriate names for everything once you enter the external name for the property. If you want to accept the default names, the only things you need to specify are the external name, the type, and whether the property is a member variable or uses `Get` and `Set` methods. If you choose a stock property from the list of available stock properties, the rest of the elements are automatically specified for you. Once you specify all of this information, the Class Wizard adds all of the necessary code and variables to your control project.

Methods

Methods are functions in the control that can be called by the container application. These functions are made available to other applications through the `IDispatch` interface, which we discussed on Day 9. Because of the way the `IDispatch` works in calling the methods in a control, the variables passed to the method have to be packaged in a structure that is passed to the control. This structure is machine independent so that it doesn't matter whether your control is running with Windows 95/98 on an Intel Pentium II or on a Windows NT with a MIPS or Alpha processor; the structure will look the same. It is the responsibility of each side of the function call to convert the parameters as necessary to fit them into the structure correctly or to extract them from the structure. This process of packaging the method parameters is called marshaling.

When you add a new method to your control through the Class Wizard on the Automation tab, the Class Wizard adds all of the necessary code to perform the marshaling of the parameters, as well as all other supporting functionality, including building the `IDispatch` interface and table.

When you add a new method to your control through the Class Wizard, you are asked to provide the external name for the method called by the container application. Your method will get a default internal name, which you can override by entering your own internal name. Other aspects of your control methods that you have to specify are the method's return type and the parameters for the method. Once you finish entering all this information, the Class Wizard adds all the necessary code to the control.

Events

Events are notification messages that are sent from the control to the container application. They are intended to notify the application that a certain event has happened, and the application can take action on that event if desirable. You can trigger two types of events from your control, stock or custom events. Stock events are implemented by the ActiveX control development kit and are available as function calls within the control. These stock events enable you to trigger events in the container application for mouse or keyboard events, errors, or state changes.

Along with the stock events, you can add your own custom events to be triggered in the container application. These events should be related to the specific functionality of your control. You can specify arguments to be passed with the event to the container application so that the application can have the data it needs for reacting to the event message.

When you need to trigger any of these events, all you do is call the internal event function that fires the event, passing all the necessary parameters to the function. The Class Wizard will have added all of the necessary code to trigger the event message from the internal function call.

Events are one of the three elements that you do not add to your controls through the Automation tab in the Class Wizard. Events are added through the ActiveX Events tab in the Class Wizard.

Creating an ActiveX Control

The ActiveX control that you will build as the example today is the squiggle drawing module that you packaged as a library module and then as DLLs on Day 16, "Creating Your Own Classes and Modules," and Day 17, "Sharing Your Functionality with Other Applications—Creating DLLs." In converting this module into an ActiveX control,

you'll expose the maximum number of squiggles that the control will draw, as well as the maximum length of the squiggles, as properties that the container application can set. Every time the control is clicked, you'll program it to create a new squiggle drawing. You'll also add a method to load a squiggle drawing into the control that was created with the previous versions of the squiggle module. Finally, you'll have the control fire an event to let the container application know that the control has loaded the drawing.

Building the Control Shell

You've probably noticed by now that one of the options on the new project dialog is an MFC ActiveX Control Wizard. This is another project wizard just like the AppWizard for creating application and DLL projects. You can use it to build a shell for any ActiveX controls that you want to build. It will create all of the necessary files and configure the project so that the compiler will build an ActiveX control when you compile.

When you start the Control Wizard, you are asked some simple questions about your control project, such as how many controls will be in the project and whether the controls will have runtime licenses.

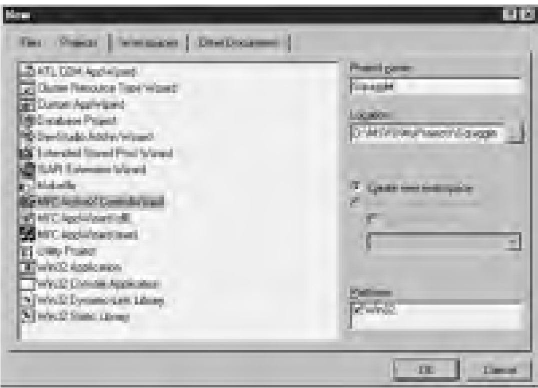
Note

> Runtime licenses are a means of making sure that the user of your control has purchased a license to use the control. Controls developed for selling to developers often have runtime licenses. The license prevents use of a control by users who haven't paid for it. When you use the control in an application, either the runtime license for the control is installed in the user's registry by the install routine or the runtime license is compiled into the application. These means prevent someone from using the control to build new applications.

In the second step of the Control Wizard, the questions get a little more involved but are still fairly easy to answer. In this step, you can click the Edit Names button to provide the control with descriptive names for the user. At the bottom of the Control Wizard, you'll find a combo box that lists a number of window classes that you can subclass in your control. If you want to create a special edit box that performs some special edits on anything the user types into the box, you choose `EDIT` from the list of window classes in the drop-down portion of this combo box. If you choose to click the Advanced button, the questions about your project require a fairly thorough understanding of ActiveX controls.

To begin the sample control project today, start a new project, selecting the MFC ActiveX Control Wizard and giving the project a suitable name, such as `Squiggle`, as shown in Figure 19.1.

FIGURE 19.1.
Starting an ActiveX control project.



Leave all the options with their default settings in the first Control Wizard step because you'll create only a single control today, and you won't need to include any runtime licensing. On the second Control Wizard step, click the Edit Names button and make sure that the type name is sufficiently descriptive of the control. Click OK to approve the names, returning to the second Control Wizard step. If you had specified in the first step that you were creating multiple controls, then you would choose each control in the drop-down list beside the Edit Names button, specifying the names for each individual control in the project. You can leave the rest of the options in the Control Wizard at their default settings for this sample project.

Modifying the `CModArt` Class

Once you create the control shell, copy the `Line` and `ModArt` files from the library module project directory, the project you built on Day 16. Load all four of these files into the control project, adding the `CLine` and `CModArt` classes to the project.

The primary changes that you need to make to the `CModArt` class for your control is setting the maximum number of squiggles and length of squiggles variables that can be exposed as control properties. To be able to implement this, you'll add two member variables to the `CModArt` class, one to control the length of the squiggles and the other to control the number of squiggles. Add these two variables to the `CModArt` class as in Table 19.1.

TABLE 19.1. MEMBER VARIABLES FOR `CModArt` CLASS.

Name	*Type*	*Access*
`m_iLength`	`int`	Private
`m_iSegments`	`int`	Private

You need to provide a way for these variables to be retrieved and updated from the exposed properties. This means that you'll need functions for getting the current value, and for setting the new value, for each of these variables. To add these functions for the `m_iLength` variable, add a member function to the `CModArt` class, specifying the type as `int`, the declaration as `GetLength`, and the access as public. Edit the function with the code in Listing 19.1.

LISTING 19.1. THE `CModArt` `GetLength` FUNCTION.

```
1: int CModArt::GetLength()
2: {
3:     // Return the current value for the m_iLength variable
4:     return m_iLength;
5: }
```

Next, add another member function to the `CModArt` class, specifying the function type as void, the declaration as `SetLength(int iLength)`, and the access as public. Edit this function, adding the code in Listing 19.2.

LISTING 19.2. THE `CModArt` `SetLength` FUNCTION.

```
1: void CModArt::SetLength(int iLength)
2: {
3:     // Set the current value for the m_iLength variable
4:     m_iLength = iLength;
5: }
```


Add the same two functions for the `m_iSegments` variable so that it can also be exposed as a property of the control.

Now that you have made these two properties available for the control, you'll make sure that they have been initialized to reasonable values before the control is used. To initialize these values, modify the `CModArt` constructor as in Listing 19.3.

LISTING 19.3. THE MODIFIED `CModArt` CONSTRUCTOR.

```
1: CModArt::CModArt()
2: {
3:     // Initialize the random number generator
4:     srand((unsigned)time(NULL));
5:     // Initialize the property variables
6:     m_iLength = 200;
7:     m_iSegments = 50;
8: }
```

Finally, you'll modify the two function that create the squiggle drawings so that they use these variables instead of the hard-coded values that they currently use. To modify the `NewDrawing` function, replace the maximum number of squiggles in line 7 with the variable `m_iSegments`, as in Listing 19.4.

LISTING 19.4. THE MODIFIED `CModArt NewDrawing` FUNCTION.

```
 1: void CModArt::NewDrawing()
 2: {
 3:     int lNumLines;
 4:     int lCurLine;
 5:
 6:     // Determine how many lines to create
 7:     lNumLines = rand() % m_iSegments;
 8:     // Are there any lines to create?
 9:     if (lNumLines > 0)
10:     {
11:         // Loop through the number of lines
12:         for (lCurLine = 0; lCurLine < lNumLines; lCurLine++)
13:         {
14:             // Create the new line
15:             NewLine();
16:         }
17:     }
18: }
```

Finally, replace the maximum length of each squiggle with the `m_iLength` variable on line 20 in the `NewLine` function, as in Listing 19.5.

LISTING 19.5. THE MODIFIED `CModArt NewLine` FUNCTION.

```
 1: void CModArt::NewLine()
 2: {
 3:     int lNumLines;
.
.
.
18:
19:     // Determine the number of parts to this squiggle
20:     lNumLines = rand() % m_iLength;
21:     // Are there any parts to this squiggle?
.
.
.
67: }
```

You have made all of the necessary modifications to the `CModArt` and `CLine` classes for your ActiveX control. Now you have to add an instance of the `CModArt` class to the control class as a member variable. Add a new member variable to the control class, `CSquiggleCtrl`, specifying its type as `CModArt`, its name as `m_maDrawing`, and its access as private. You also need to include the header file for the `CModArt` class in the control class source code file, so open this file, scroll to the top of the file, and add an include statement for the `ModArt.h` file, as in Listing 19.6.

LISTING 19.6. THE `CSquiggleCtrl` INCLUDES.

```
1: // SquiggleCtl.cpp : Implementation of the CSquiggleCtrl ActiveX
Control class.
2:
3: #include "stdafx.h"
4: #include "Squiggle.h"
5: #include "SquiggleCtl.h"
6: #include "SquigglePpg.h"
7: #include "ModArt.h"
```

Adding Properties

Because the two variables that you added to the `CModArt` class are not variables of the control class (`CSquiggleCtrl`), you will probably want to add `Get` and `Set` methods to set and retrieve the property value. If these two variables were members of the control class, you could add them through the Class Wizard as member variables. You would still know when and if the variables had been changed because you would have a notification method in the control class that would be called when the property values are changed. However, because they are members of an internal class, you'll want to exercise a little more control over their values.

Even if the variables that you want to expose are member variables of the control class, you might still want to use the `Get` and `Set` methods for accessing the variables as control properties. Using the `Get` and `Set` methods allow you to add validation on the new value for the properties so that you can make certain that the container application is setting an appropriate value to the property.

To add these properties to your control, open the Class Wizard and select the Automation tab, as in Figure 19.2. Click on the Add Property button to add the first property. In the Add Property dialog, enter the external name that you want your property to have, such

as `SquiggleLength`, and specify the type as `short` (the `int` type is not available, only `short` and `long`). Click the `Get/Set` methods radio button, and the dialog enters function names for these two methods, as in Figure 19.3. Click OK to add this property.

FIGURE 19.2.

The Class Wizard Automation tab.

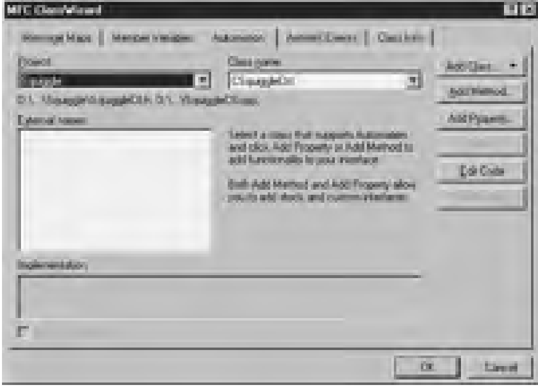
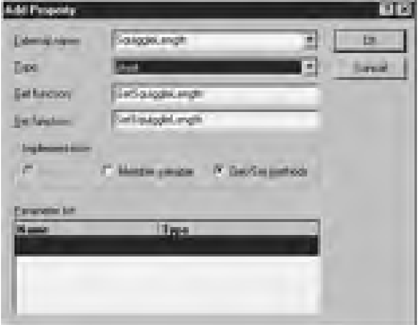


FIGURE 19.3.

The Add Property dialog.



Click the Edit Code button to add the code for the `Get` and `Set` methods. In each method, you'll call the `Get` and `Set` functions that you added to the `CModArt` class to control access to the length variable. Edit these two methods as shown in Listing 19.7.

LISTING 19.7. THE `CSquiggleCtrl Get/SetSquiggleLength` FUNCTIONS.

```
1:  short CSquiggleCtrl::GetSquiggleLength()
2:  {
3:      // TODO: Add your property handler here
4:      // Return the result from the GetLength function
5:      return m_maDrawing.GetLength();
6:  }
7:
```

```
 8: void CSquiggleCtrl::SetSquiggleLength(short nNewValue)
 9: {
10:     // TODO: Add your property handler here
11:     // Set the new length value
12:     m_maDrawing.SetLength(nNewValue);
13:     SetModifiedFlag();
14: }
```

Add another property for the number of squiggles in a drawing by following the same steps, substituting an appropriate property name, such as `NumberSquiggles`.

One last property you might want to add to your control is a boolean property that the container application could use to keep the control from creating any new drawings and to keep the current drawing visible. Add a new property through the Class Wizard, giving it a suitable name such as `KeepCurrentDrawing`, and specify the type as `BOOL`. Leave this property set as a member variable and click OK. The Class Wizard automatically adds the variable to the control class, along with all of the necessary code to maintain the variable.

Designing and Building the Property Page

You need to provide a property page with your control that developers can use when they are working with your control. This property page will provide the users with a means of setting the properties of the control, even if their own development tools do not provide them with a facility to get to these properties in any way other than with code.

Adding a property page to your control is pretty easy. If you select the Resources view tab in the workspace and expand the dialog folder, you'll see a dialog for your control's property page already in the folder. Open this dialog, and you'll find that it's a standard dialog window that you can design using the standard controls available in the dialog designer. To design the property page for your sample control, lay out the property page dialog as shown in Figure 19.4, using the property settings in Table 19.2.

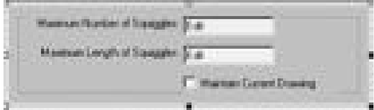


FIGURE 19.4.
The control property page layout.

TABLE 19.2. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	ID	`IDC_STATIC`
	Caption	`Maximum Number of Squiggles:`
Edit Box	ID	`IDC_ENBRSQUIG`
Static Text	ID	`IDC_STATIC`
	Caption	`Maximum Length of Squiggles:`
Edit Box	ID	`IDC_ELENSQUIG`
Check Box	ID	`IDC_CMAINTDRAW`
	Caption	`Maintain Current Drawing`

Once you add all the controls and specify their properties, open the Class Wizard to add variables for these controls. When you add a variable to one of the controls on the property page dialog, you'll notice an additional combo box on the Add Member Variable dialog. This new combo box is for the external name of the property that the variable should be tied to in the control. The drop-down list on this combo box is a list of all of the standard properties that you might want to tie the property page control to, but if you are tying it to a custom property, you have to enter the property name yourself, as shown in Figure 19.5.

FIGURE 19.5.
The Add Member Variable dialog.

Add variables to the controls on the property page for your control, tying them to the control's properties, as specified in Table 19.3.

TABLE 19.3. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*	*Property*
`IDC_CMAINTDRAW`	`m_bKeepDrawing`	`Value`	`BOOL`	`KeepCurrentDrawing`
`IDC_ELENSQUIG`	`m_iLenSquig`	`Value`	`int`	`SquiggleLength`
`IDC_ENBRSQUIG`	`m_iNbrSquiggles`	`Value`	`int`	`NumberSquiggles`

Click the OK button to add all these variables to the control property page class.

Adding Basic Control Functionality

The basic functionality that your control needs is the ability to respond to mouse clicks by generating a new drawing. To control this behavior, you'll add a second boolean variable to the control class so that the `OnDraw` function knows that a mouse click has been triggered. The easiest place to get the drawing area of the control is the `OnDraw` function, so this is where the new drawing needs to be generated. Do you want the control to generate a new drawing every time the user moves the application using your control in front of another application? Probably not. You will most likely want a greater amount of control over the behavior of the control, so it makes sense to add this second boolean variable. Add a member variable to the control class (`CSquiggleCtrl`), specifying the variable type as `BOOL`, the variable name as `m_bGenNewDrawing`, and the variables access as private.

Before you start adding the code to perform all the various tasks, it's important that you initialize all the member variables in the control class. This consists of the member variable property, `m_keepCurrentDrawing`, and the member variable that you just added, `m_bGenNewDrawing`. You'll want your control to generate a new drawing right off the bat, and you probably don't want it to maintain any drawings, unless the container application explicitly specifies that a drawing is to be maintained. You'll set these two variables accordingly in the control class constructor, as shown in Listing 19.8.

LISTING 19.8. THE `CSquiggleCtrl` CONSTRUCTOR.

```
1: CSquiggleCtrl::CSquiggleCtrl()
2: {
3:     InitializeIIDs(&IID_DSquiggle, &IID_DSquiggleEvents);
4:
5:     // TODO: Initialize your control's instance data here.
6:     // Initialize the variables
7:     m_bGenNewDrawing = TRUE;
8:     m_keepCurrentDrawing = FALSE;
9: }
```


Next, you'll add the code to generate and display the squiggle drawings. The place to add this functionality is the `OnDraw` function in the control class. This function is called every time that the control needs to draw itself, whether it was hidden or something triggered the redrawing by calling the `Invalidate` function on the control. Once in the `OnDraw` function, you'll determine whether you need to generate a new drawing or just draw the existing drawing. Another thing to keep in mind is that you are responsible for

drawing the entire area that the control occupies. This means that you need to draw the background of the squiggle drawing, or else the squiggles will be drawn on top of whatever was displayed in that same spot on the screen. (Who knows? That might be the effect you are looking for.) To add this functionality to your control, edit the `OnDraw` function in the control class, adding the code in Listing 19.9.

LISTING 19.9. THE `CSquiggleCtrl` `OnDraw` FUNCTION.

```
 1: void CSquiggleCtrl::OnDraw(
 2:             CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
 3: {
 4:     // TODO: Replace the following code with your own drawing code.
 5:     //pdc->FillRect(rcBounds, CBrush::FromHandle((HBRUSH)
            ➥GetStockObject(WHITE_BRUSH)));
 6:     //pdc->Ellipse(rcBounds);
 7:     // Do we need to generate a new drawing?
 8:     if (m_bGenNewDrawing)
 9:     {
10:         // Set the drawing area for the new drawing
11:         m_maDrawing.SetRect(rcBounds);
12:         // Clear out the old drawing
13:         m_maDrawing.ClearDrawing();
14:         // Generate the new drawing
15:         m_maDrawing.NewDrawing();
16:         // Reset the control flag
17:         m_bGenNewDrawing = FALSE;
18:     }
19:     // Fill in the background
20:     pdc->FillRect(rcBounds,
21:         CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
22:     // Draw the squiggle drawing
23:     m_maDrawing.Draw(pdc);
24: }
```

Finally, you'll trigger the control to generate a new drawing whenever the control is clicked. This requires adding an event handler for the control's `OnClick` event. First, however, you'll add a stock method to the control to make sure that it receives the `OnClick` event message. To add this stock method, open the Class Wizard and select the Automation tab. Add a new method to the control class, selecting the `DoClick` method from the drop-down list of stock methods that can be added to your control, as shown in Figure 19.6. Click the OK button to add the method to your control, and then select the Message Maps tab in the Class Wizard. Select the `OnClick` event message from the list of available event messages, and add a function to handle this event message. Edit the code for the `OnClick` event handler, adding the code in Listing 19.10.

FIGURE 19.6.
The Add Method dialog.

LISTING 19.10. THE `CSquiggleCtrl` `OnClick` FUNCTION.

```
 1: void CSquiggleCtrl::OnClick(USHORT iButton)
 2: {
 3:     // TODO: Add your specialized code here and/or call the base class
 4:     // Can we generate a new drawing?
 5:     if (!m_keepCurrentDrawing)
 6:     {
 7:         // Set the flag so a new drawing will be generated
 8:         m_bGenNewDrawing = TRUE;
 9:         // Invalidate the control to trigger the OnDraw function
10:         Invalidate();
11:     }
12:     COleControl::OnClick(iButton);
13: }
```


In the `OnClick` function, you check to see whether you could generate a new drawing or maintain the current drawing. If you could generate a new drawing, you set the `m_bGenNewDrawing` flag to `TRUE` and invalidated the control, which triggers the `OnDraw` function.

Adding Methods

Remember the functionality that you are going to give your control: One of the functions is loading a squiggle drawing created with the version of the Squiggle module that you created on Day 16. To add this functionality, you'll add a method to the control that the container application can call to pass a filename to be loaded. You've already added one method to your application, a stock method. Adding a custom method is similar, but you have to provide a little more information to the Add Method dialog.

In the method to load an existing drawing, you'll create a `CFile` object for the filename that was passed as a parameter. The `CFile` constructor will take the filename and the flag `CFile::modeRead` to let it know that you are opening the file for reading only. Once you

create the CFile object, you'll create a CArchive object to read the file. The CArchive constructor will take the CFile object that you just created and the CArchive::load flag to tell it that it needs to load the file. At this point, you can pass the CArchive object to the drawing object's Serialize function and let it read and load the drawing. Once the drawing is loaded, you need to display the drawing by invalidating the control. Before you invalidate the control, you probably want to make sure that the m_bGenNewDrawing flag is set to FALSE so that the drawing you just loaded won't be overwritten.

To add this functionality to your control, open the Class Wizard and select the Automation tab. Click the Add Method button to add a custom method. Enter the external method name in the first combo box; in this case, call it LoadDrawing. The internal name will automatically be generated based on the external name you entered. Next, specify the return type as BOOL so that you can let the container application know whether you were able to load the drawing. Finally, add a single parameter to the parameter list, giving it a name such as sFileName and specifying its type as LPCTSTR (the CString type is not available, but the LPCTSTR type is compatible), as shown in Figure 19.7. Click the OK button to add the method to your control. Once you add the method, click the Edit Code button to edit the method, adding the code in Listing 19.11.

FIGURE 19.7.

The Add custom Method dialog.

LISTING 19.11. THE CSquiggleCtrl LoadDrawing FUNCTION.

```
 1: BOOL CSquiggleCtrl::LoadDrawing(LPCTSTR sFileName)
 2: {
 3:     // TODO: Add your dispatch handler code here
 4:     try
 5:     {
 6:         // Create a CFile object
 7:         CFile lFile(sFileName, CFile::modeRead);
 8:         // Create a CArchive object to load the file
 9:         CArchive lArchive(&lFile, CArchive::load);
10:         // Load the file
```

```
11:         m_maDrawing.Serialize(lArchive);
12:         // Make sure that the loaded drawing won't be overwritten
13:         m_bGenNewDrawing = FALSE;
14:         // Draw the loaded drawing
15:         Invalidate();
16:     }
17:     catch (CFileException err)
18:     {
19:         return FALSE;
20:     }
21:     return TRUE;
22: }
```

Adding Events

The final part of building your control is adding the events that your control will trigger in the container application. When using your control, the user will be able to add code to be triggered on these events. Adding these events to your control is done through the ActiveX Events tab of the Class Wizard. If you want to add a stock event to be triggered by your control, then you just click the Add Event button and select a stock event from the drop-down list of stock events. If you need to add a custom event to your control, then in the Add Event dialog, instead of selecting a stock event, you enter the name of your custom event. At the bottom of the Add Event dialog is an area for adding parameters that you can pass from your control to the container application with the event.

For the sample control, you'll add one event, a custom event to let the application know that the drawing file specified has been loaded. To add this event, open the Class Wizard and select the ActiveX Events tab, as shown in Figure 19.8. Click the Add Event button to add the event. Enter the name for your custom event, `FileLoaded`. You'll notice that the Add Event dialog automatically builds an internal name for the event, in this case, `FireFileLoaded`, as shown in Figure 19.9. This internal name is the name for the function that you need to call in your code when you want to trigger this event. Click the OK button to add this event. To add a stock event, select the desired stock event from the drop-down list of stock events, and click the OK button to add this second event.

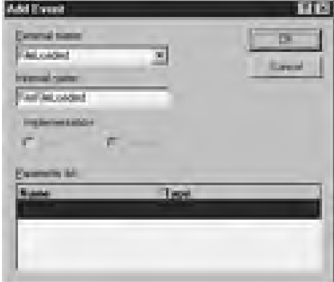
Now that you've added your event to your control, you need to make the necessary changes to the code to trigger this event at the appropriate places. You'll trigger your event at the end of your `LoadDrawing` function, assuming that you are able to load the drawing correctly. Add this additional functionality to the `LoadDrawing` function, as shown in line 17 of Listing 19.12.

FIGURE 19.8.

The ActiveX Events tab of the Class Wizard.

FIGURE 19.9.

The Add Event dialog.



LISTING 19.12. THE MODIFIED `CSquiggleCtrl LoadDrawing` FUNCTION.

```
 1: BOOL CSquiggleCtrl::LoadDrawing(LPCTSTR sFileName)
 2: {
 3:     // TODO: Add your dispatch handler code here
 4:     try
 5:     {
 6:         // Create a CFile object
 7:         CFile lFile(sFileName, CFile::modeRead);
 8:         // Create a CArchive object to load the file
 9:         CArchive lArchive(&lFile, CArchive::load);
10:         // Load the file
11:         m_maDrawing.Serialize(lArchive);
12:         // Make sure that the loaded drawing won't be overwritten
13:         m_bGenNewDrawing = FALSE;
14:         // Draw the loaded drawing
15:         Invalidate();
16:         // Fire the FileLoaded event
17:         FireFileLoaded();
18:     }
19:     catch (CFileException err)
```

```
20:     {
21:         return FALSE;
22:     }
23:     return TRUE;
24: }
```

Testing the Control

Now you are ready to compile and begin testing your control. Before you run to the store to pick up a copy of Visual Basic, you already have a tool just for testing ActiveX controls. On the Tools menu is one entry labeled ActiveX Control Test Container. This is a utility that is designed specifically for testing ActiveX controls that you have built. Once you compile your control, run the ActiveX Control Test Container to test your control.

Tip

> If Visual C++ is unable to register your control, but is able to compile it, you might need to register your control yourself. You can do this by selecting Tools|Register Control from the menu. This will register the compiled control in the Registry database.

When you first start the test container, you see a blank area where your control will appear. You need to insert your control into this container area by selecting Edit|Insert New Control. This will open the Insert Control dialog, as shown in Figure 19.10. Select your control from the list of available controls and click the OK button to add your control to the container area, as shown in Figure 19.11.

FIGURE 19.10.
The Insert Control dialog.

Now that you have your control loaded into the test container, you can play with it, resize it, click it, and check when it generates a new drawing and when it just redraws the existing drawing. If you trigger any events for your control, you'll see the event that your control fired in the bottom pane of the test container so that you can watch as each of the events you added to your control are triggered.

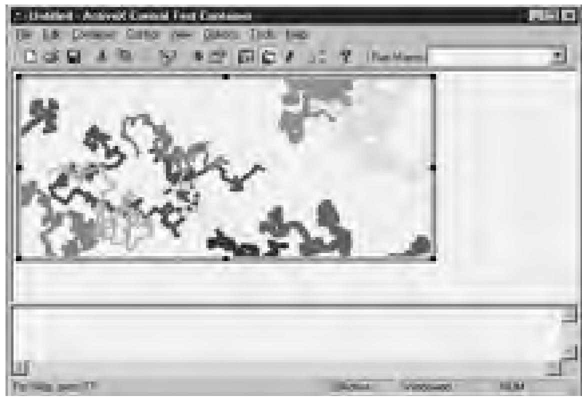


FIGURE 19.11.

The squiggle control in the test container.

With your control selected, if you select Edit|Properties from the menu, you'll open the property page that you designed for your control, allowing you to modify the various properties of the control so that you can see whether they work correctly, as shown in Figure 19.12.

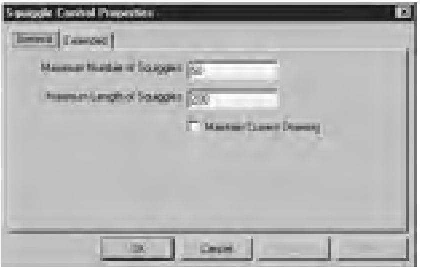


FIGURE 19.12.

The Squiggle Control Properties page.

Finally, to test the methods that you added to your control, select Control|Invoke Methods. This opens the Invoke Methods dialog, as shown in Figure 19.13. In here, you can select from the list of available methods in your control, entering each of the parameters required for the methods, and then click the Invoke button to call that method. You can watch as your methods are called and your control responds.

FIGURE 19.13.
The Invoke Methods dialog.

Summary

Today, you learned how you can use the tools and wizards in Visual C++ to build ActiveX controls with little effort on your part. You learned how you can create the shell of the control project using the Control Wizard. You also learned how you can use the Class Wizard to add properties, methods, and events to your control. You saw how you can design a property page for your control and how you can use the Class Wizard to attach the controls on this dialog to the properties you defined for your control without having to add any code. Finally, you learned how you can use the ActiveX Control Test Container to test your control, triggering all the functionality by using the tools of this utility.

Q&A

Q How do I change the icon that appears in the toolbox for my control?

A In the Resource View tab of the workspace pane, open the Bitmap folder. You should find a single bitmap in this folder. This image is displayed in the toolbox for your control when you add it to a Visual C++ or Visual Basic project. You should edit this bitmap so that it displays the image that you want to represent your control.

Q Why does my control have an About box?

A If you are building ActiveX controls that will be used by other developers, whether you sell the control or give it away, you probably want to include some way of indicating that you wrote the control, and that you, or your employer, owns the copyright on the control. This acts as a legal identification on the control so that whoever obtains your control cannot turn around and sell it as his creation.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What are the three aspects of a control that are visible to the container application?
2. Why do you need to design a property page for your control?
3. What are the four types of properties that a control might have?
4. What happens to the parameters that are passed to the methods of a control?
5. What tool can you use to test your controls?

Exercises

1. Add a method to your control to enable the container application to trigger the generation of a new squiggle drawing.
2. Add a method to your control to save a squiggle drawing. Use the `CFile::modeWrite` and `CArchive::store` flags when creating the `CFile` and `CArchive` objects.

WEEK 3

DAY 20

Internet Applications and Network Communications

Thanks in part to the explosion in popularity of the Internet, more applications have the ability to communicate with other applications over networks, including the Internet. With Microsoft building networking capabilities into its operating systems, starting with Windows NT and Windows 95, these capabilities are becoming commonplace in all sorts of applications.

Some applications perform simple networking tasks such as checking with a Web site to see whether there are any updates to the program and giving the user the option of updating her copy of the program. Some word processing applications will format documents as Web pages, giving the user the option of loading the pages onto the Web server. You've got computer games that allow the user to play against another person halfway around the world instead of just competing against the game itself.

Applications can have any number of networking functions, and they all are built around the Winsock interface. If you know and understand how to program using the Winsock interface and the MFC Winsock classes, this entire

realm of application programming is open to you, expanding your programming options considerably. Today, you will learn

- How applications use the Winsock interface to perform network communications between two or more computers.
- The difference between a client and a server application and the role each plays in establishing a communications link.
- How the MFC Winsock classes simplify the process of writing Internet applications.
- How you can create your own Winsock class, descended from the MFC Winsock classes, to easily build an event-driven, networking application.

How Do Network Communications Work?

Most applications that communicate over a network, whether it's the Internet or a small office network, use the same principles and functionality to perform their communication. One application sits on a computer, waiting for another application to open a communication connection. This application is "listening" for this connection request, much like you listen for the phone to ring if you are expecting someone to call.

Meanwhile, another application, most likely on another computer (but not necessarily), tries to connect to the first application. This attempt to open a connection is similar to calling someone on the telephone. You dial the number and hope that the other person is listening for the phone on the other end. As the person making the call, you have to know the phone number of the person you are calling. If you don't know the phone number, you can look it up using the person's name. Likewise, the application trying to connect to the first application has to know the network location, or address, of the first application.

Once the connection is made between the two applications, messages can pass back and forth between the two applications, much like you can talk to the person on the other end of the phone. This connection is a two-way communications channel, with both sides sending information, as shown in Figure 20.1.

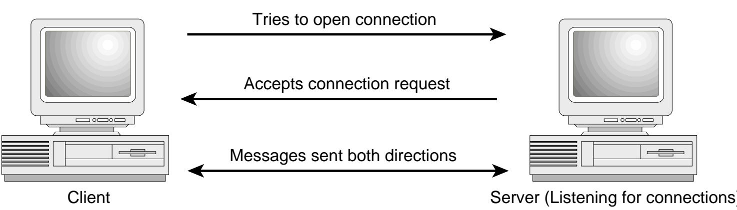


FIGURE 20.1.
The basic socket connection process.

Finally, once one or both sides have finished their sides of the conversation, the connection is closed, much like you hang up the phone after you have finished talking to the person you called. Once the connection is closed from either side, the other side can detect it and close its side, just like you can tell if the person on the other end of the phone has hung up on you or if you've been disconnected by some other means. This is a basic explanation of how network communications work between two or more applications.

Note

This is a basic description of how network communications work with the TCP/IP network protocol, which is the primary network protocol over the Internet. Many other network protocols use a subtle variation on this description. Other protocols, such as the UDP protocol, are more like radio broadcasts, where there is no connection between the two applications; one sends messages, and the other is responsible for making sure that it receives all of the messages. These protocols are more involved than we have the luxury to discuss today. If you want to learn more about network protocols and how they work, many books cover this one topic and look at the various Internet applications and how they communicate over the connections they establish.

Sockets, Ports, and Addresses

The basic object used by applications to perform most network communications is called a socket. Sockets were first developed on UNIX at the University of California at Berkley. Sockets were designed so that most network communications between applications could be performed in the same way that these same applications would read and write files. Sockets have progressed quite a bit since then, but the basics of how they work are still the same.

During the days of Windows 3.*x*, before networking was built into the Windows operating system, you could buy the network protocols required for network communications from numerous different companies. Each of these companies had a slightly different way that an application performed network communications. As a result, any applications that performed network communications had a list of the different networking software that the application would work with. Many application developers were not happy with this situation. As a result, all the networking companies, including Microsoft, got together and developed the Winsock (Windows Sockets) API. This provided all application developers with a consistent API to perform all network communications, regardless of the networking software used.

When you want to read or write a file, you must use a file object to point to the file. Although this was hidden from you in most of the Visual C++ applications so far, with the ActiveX control you created yesterday, you had to work through the steps of creating the file object for reading and writing. A socket is similar; it is an object used to read and write messages that travel between applications.

Making a socket connection to another application does require a different set of information than opening a file. To open a file, you need to know the file's name and location. To open a socket connection, you need to know the computer on which the other application is running and the port on which it's listening. A port is like a phone extension, and the computer address is like the phone number. If you call someone at a large office building, you may dial the main office number, but then you need to specify the extension number. Likewise, ports are used to route network communications (see Figure 20.2). As with the phone number, there are means of looking up the port number, if you don't already know what it is, but this requires your computer to be configured with the information about which port the connecting application is listening on. If you specify the wrong computer address or port number, you may get a connection to a different application; with making the phone call, someone other than the person you called may answer the phone call. You also may not get an answer at all if there is no application listening at the other end.

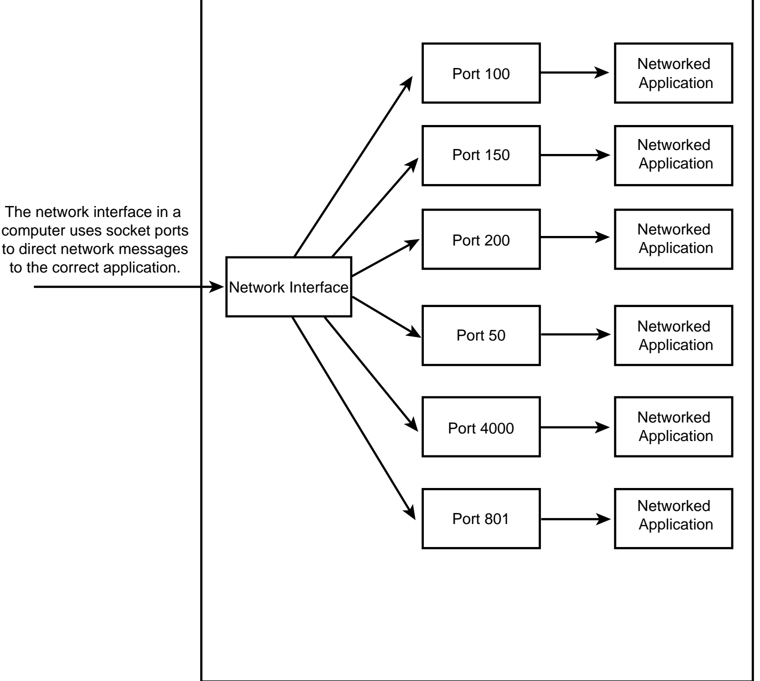
Only one application may be listening on any specific port on a single computer. Although numerous applications may listen for connection requests on a single computer at the same time, each of these applications must listen on a different port.

Creating a Socket

When you build applications with Visual C++, you can use the MFC Winsock classes to add network communications capabilities with relative ease. The base class, `CAsyncSocket`, provides complete, event-driven socket communications. You can create your own descendent socket class that captures and responds to each of these events.

This discussion of socket communications assumes that you check the AppWizard option for adding support for Windows Sockets. This adds supporting functionality to the application that is not discussed here.

FIGURE 20.2.
Ports are used to route network communications to the correct application.



To create a socket that you can use in your application, the first thing you need to do is declare a variable of `CAsyncSocket` (or your descendent class) as a class member for one of the main application classes:

```
class CMyDlg : public CDialog
{
.
.
.
private:
    CAsyncSocket m_sMySocket;
};
```

Before you can begin using the socket object, you must call its `Create` method. This actually creates the socket and prepares it for use. How you call the `Create` method depends on how you will be using the socket. If you will be using the socket to connect to another application, as the one placing the call (the client), then you do not need to pass any parameters to the `Create` method:

```
if (m_sMySocket.Create())
{
    // Continue on
```

```
}
else
    // Perform error handling here
```

However, if the socket is going to be listening for another application to connect to it, waiting for the call (the server), then you need to pass at least the port number on which the socket should be listening:

```
if (m_sMySocket.Create(4000))
{
    // Continue on
}
else
    // Perform error handling here
```

You can include other parameters in the `Create` method call, such as the type of socket to create, the events that the socket should respond to, and the address that the socket should listen on (in case the computer has more than one network card). All these options require a more thorough understanding of sockets than we'll be able to cover today.

Making a Connection

Once you create a socket, you are ready to open a connection with it. Three steps go along with opening a single connection. Two of these steps take place on the server, the application listing for the connection, and the third step takes place on the client, the one making the call.

For the client, opening the connection is a simple matter of calling the `Connect` method. The client has to pass two parameters to the `Connect` method: the computer name, or network address, and the port of the application to connect to. The `Connect` method could be used in the following two ways:

```
if (m_sMySocket.Connect("thatcomputer.com", 4000))
{
    // Continue on
}
else
    // Perform error handling here
```

The second form is

```
if (m_sMySocket.Connect("178.1.25.82", 4000))
{
    // Continue on
}
else
    // Perform error handling here
```

Once the connection is made, an event is triggered to let your application know that it is connected or that there were problems and the connection couldn't be made. (I'll cover how these events work in the section "Socket Events," later in this chapter.)

For the server, or listening, side of the connection, the application first must tell the socket to listen for incoming connections by calling the `Listen` method. The `Listen` method takes only a single argument, which you do not need to supply. This parameter specifies the number of pending connections that can be queued, waiting for the connection to be completed. By default this value is `5`, which is the maximum. The `Listen` method can be called as follows:

```
if (m_sMySocket.Listen())
{
    // Continue on
}
else
    // Perform error handling here
```

Whenever another application is trying to connect to the listening application, an event is triggered to let the application know that the connection request is there. The listening application must accept the connection request by calling the `Accept` method. This method requires the use of a second `CAsyncSocket` variable, which is connected to the other application. Once a socket is placed into listen mode, it stays in listen mode. Whenever connection requests are received, the listening socket creates another socket, which is connected to the other application. This second socket should not have the `Create` method called for it because the `Accept` method creates the socket. You call the `Accept` method as follows:

```
if (m_sMySocket.Accept(m_sMySecondSocket))
{
    // Continue on
}
else
    // Perform error handling here
```

At this point, the connecting application is connected to the second socket on the listening application.

Sending and Receiving Messages

Sending and receiving message through a socket connection gets slightly involved. Because you can use sockets to send any kind of data, and they don't care what the data is, the functions to send and receive data expect to be passed a pointer to a generic buffer. For sending data, this buffer should contain the data to be sent. For receiving data, this buffer will have the received data copied into it. As long as you are sending

and receiving strings and text, you can use fairly simple conversions to and from `CStrings` with these buffers.

To send a message through a socket connection, you use the `Send` method. This method requires two parameters and has a third, optional parameter that can be used to control how the message is sent. The first parameter is a pointer to the buffer that contains the data to be sent. If your message is in a `CString` variable, you can use the `LPCTSTR` operator to pass the `CString` variable as the buffer. The second parameter is the length of the buffer. The method returns the amount of data that was sent to the other application. If an error occurs, the `Send` function returns `SOCKET_ERROR`. You can use the `Send` method as follows:

```
CString strMyMessage;
int iLen;
int iAmtSent;
.
.
.
iLen = strMyMessage.GetLength();
iAmtSent = m_sMySocket.Send(LPCTSTR(strMyMessage), iLen);
if (iAmtSent == SOCKET_ERROR)
{
    // Do some error handling here
}
else
{
    // Everything's fine
}
```

When data is available to be received from the other application, an event is triggered on the receiving application. This lets your application know that it can receive and process the message. To get the message, the `Receive` method must be called. This method takes the same parameters as the `Send` method with a slight difference. The first parameter is a pointer to a buffer into which the message may be copied. The second parameter is the size of the buffer. This tells the socket how much data to copy (in case more is received than will fit into the buffer). Like the `Send` method, the `Receive` method will return the amount that was copied into the buffer. If an error occurs, the `Receive` method also returns `SOCKET_ERROR`. If the message your application is receiving is a text message, it can be copied directly into a `CString` variable. This allows you to use the `Receive` method as follows:

```
char *pBuf = new char[1025];
int iBufSize = 1024;
int iRcvd;
CString strRecvd;

iRcvd = m_sMySocket.Receive(pBuf, iBufSize);
```

```
if (iRcvd == SOCKET_ERROR)
{
    // Do some error handling here
}
else
{
    pBuf[iRcvd] = NULL;
    strRecvd = pBuf;
    // Continue processing the message
}
```

When receiving text messages, it's always a good idea to place a NULL in the buffer position just after the last character received, as in the preceding example. There may be garbage characters in the buffer that your application might interpret as part of the message if you don't add the NULL to truncate the string.

Closing the Connection

Once your application has finished all of its communications with the other application, it can close the connection by calling the `Close` method. The `Close` method doesn't take any parameters, and you use it as follows:

```
m_sMySocket.Close();
```

The `Close` function is one of the few `CAsyncSocket` methods that does not return any status code. For all the previous member functions that we have examined, you can capture the return value to determine if an error has occurred.

Socket Events

The primary reason that you create your own descendent class of `CAsyncSocket` is that you want to capture the events that are triggered when messages are received, connections are completed, and so on. The `CAsyncSocket` class has a series of functions that are called for each of these various events. These functions all use the same definition—the function name is the only difference—and they are intended to be overridden in descendent classes. All of these functions are declared as protected members of the `CAsyncSocket` class and probably should be declared as protected in your descendent classes. The functions all have a single integer parameter, which is an error code that

should be checked to make sure that no error has occurred. Table 20.1 lists these event functions and the events they signal.

TABLE 20.1. `CAsyncSocket` OVERRIDABLE EVENT-NOTIFICATION FUNCTIONS.

Function	*Event Description*
`OnAccept`	This function is called on a listening socket to signal that a connection request from another application is waiting to be accepted.
`OnClose`	This function is called on a socket to signal that the application on the other end of the connection has closed its socket or that the connection has been lost. This should be followed by closing the socket that received this notification.
`OnConnect`	This function is called on a socket to signal that the connection with another application has been completed and that the application can now send and receive messages through the socket.
`OnReceive`	This function is called to signal that data has been received through the socket connection and that the data is ready to be retrieved by calling the `Receive` function.
`OnSend`	This function is called to signal that the socket is ready and available for sending data. This function is called right after the connection has been completed. Usually, the other time that this function is called is when your application has passed the `Send` function more data than can be sent in a single packet. In this case, this is a signal that all of the data has been sent, and the application can send the next buffer-full of data.

Detecting Errors

Whenever any of the `CAsyncSocket` member functions return an error, either `FALSE` for most functions or `SOCKET_ERROR` on the `Send` and `Receive` functions, you can call the `GetLastError` method to get the error code. This function returns only error codes, and you have to look up the translation yourself. All the Winsock error codes are defined with constants, so you can use the constants in your code to determine the error message to display for the user, if any. You can use the `GetLastError` function as follows:

```
int iErrCode;

iErrCode = m_sMySocket.GetLastError();
switch (iErrCode)
{
case WASNOTINITIALISED:
.
.
.
}
```

Building a Networked Application

For the sample application that you will build today, you'll create a simple dialog application that can function as either the client or server in a Winsock connection. This will allow you to run two copies of the sample application, one for each end of the connection, on the same computer or to copy the application to another computer so that you can run the two copies on separate computers and see how you can pass messages across a network. Once the application has established a connection with another application, you will be able to enter text messages and send them to the other application. When the message has been sent, it will be added to a list of messages sent. Each message that is received will be copied into another list of all messages received. This will allow you to see the complete list of what is sent and received. It will also allow you to compare what one copy of the application has sent and what the other has received. (The two lists should be the same.)

Creating the Application Shell

For today's sample application, just to keep things simple, you'll create a dialog-style application. Everything that you are doing in today's application can be done in an SDI or MDI application just as easily as with a dialog-style application. By using a dialog-style application today, we are getting everything that might distract from the basic socket functionality (such as questions about whether the socket variable belongs in the document or view class, how much of the application functionality belongs in which of these two classes, and so on) away from the sample application.

To start today's sample application, create a new MFC AppWizard project, giving the project a suitable name, such as `Sock`. On the first step of the AppWizard, specify that the application will be a dialog-based application. On the second step of the AppWizard, specify that the application should include support for Windows Sockets, as in Figure 20.3. You can accept the default settings for the rest of the options in the AppWizard.

Window Layout and Startup Functionality

Once you create your application shell, you can lay out the main dialog for your application. On this dialog, you'll need a set of radio buttons to specify whether the application is running as the client or server. You'll also need a couple of edit boxes for the computer name and port that the server will be listening on. Next, you'll need a command button to start the application listening on the socket or opening the connection to the server, and a button to close the connection. You'll also need an edit box for entering the message to be sent to the other application and a button to send the message. Finally, you'll need a couple of list boxes into which you can add each of the messages sent and

received. Place all these controls on the dialog, as shown in Figure 20.4, setting all of the control properties as specified in Table 20.2.

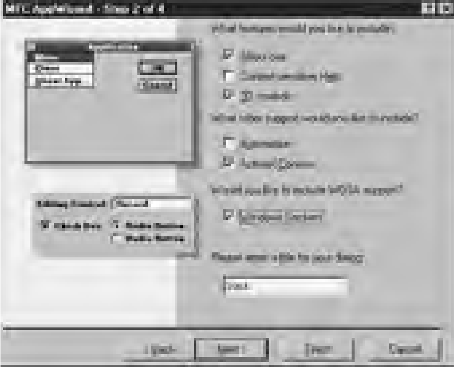


FIGURE 20.3. *Including sockets support.*

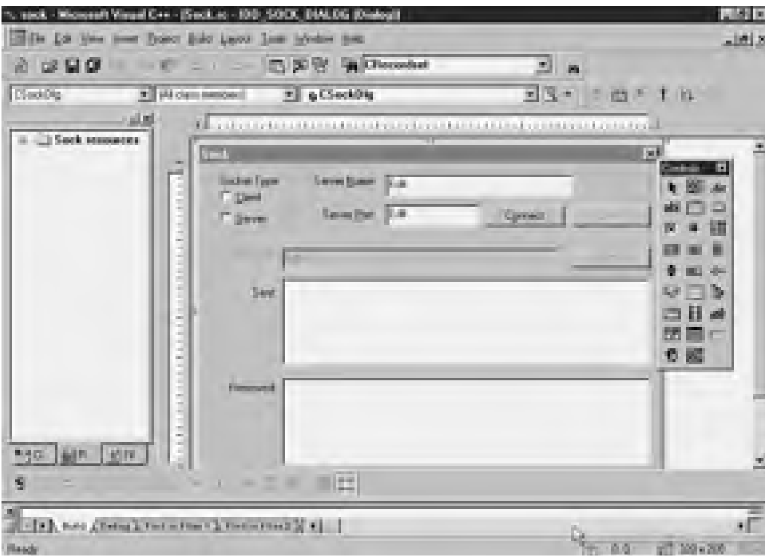


FIGURE 20.4. *The main dialog layout.*

TABLE 20.2. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Group Box	ID	`IDC_STATICTYPE`
	Caption	`Socket Type`
Radio Button	ID	`IDC_RCLIENT`
	Caption	`&Client`
	Group	`Checked`

Object	*Property*	*Setting*
Radio Button	ID	`IDC_RSERVER`
	Caption	`&Server`
Static Text	ID	`IDC_STATICNAME`
	Caption	`Server &Name:`
Edit Box	ID	`IDC_ESERVNAME`
Static Text	ID	`IDC_STATICPORT`
	Caption	`Server &Port:`
Edit Box	ID	`IDC_ESERVPORT`
Command Button	ID	`IDC_BCONNECT`
	Caption	`C&onnect`
Command Button	ID	`IDC_BCLOSE`
	Caption	`C&lose`
	Disabled	`Checked`
Static Text	ID	`IDC_STATICMSG`
	Caption	`&Message:`
	Disabled	`Checked`
Edit Box	ID	`IDC_EMSG`
	Disabled	`Checked`
Command Button	ID	`IDC_BSEND`
	Caption	`S&end`
	Disabled	`Checked`
Static Text	ID	`IDC_STATIC`
	Caption	`Sent:`
List Box	ID	`IDC_LSENT`
	Tab Stop	`Unchecked`
	Sort	`Unchecked`
	Selection	`None`
Static Text	ID	`IDC_STATIC`
	Caption	`Received:`
List Box	ID	`IDC_LRECVD`
	Tab Stop	`Unchecked`
	Sort	`Unchecked`
	Selection	`None`

Once you have the dialog designed, open the Class Wizard to attach variables to the controls on the dialog, as specified in Table 20.3.

TABLE 20.3. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
IDC_BCONNECT	m_ctlConnect	Control	CButton
IDC_EMSG	m_strMessage	Value	CString
IDC_ESERVNAME	m_strName	Value	CString
IDC_ESERVPORT	m_iPort	Value	int
IDC_LRECVD	m_ctlRecvd	Control	CListBox
IDC_LSENT	m_ctlSent	Control	CListBox
IDC_RCLIENT	m_iType	Value	int

So that you can reuse the Connect button to place the server application into listen mode, you'll add a function to the clicked event message for both radio buttons, changing the text on the command button depending on which of the two is currently selected. To add this functionality to your application, add a function to the `BN_CLICKED` event message for the `IDC_RCLIENT` control ID, naming the function `OnRType`. Add the same function to the `BN_CLICKED` event message for the `IDC_RSERVER` control ID. Edit this function, adding the code in Listing 20.1.

LISTING 20.1. THE `CSockDlg` `OnRType` FUNCTION.

```
 1: void CSockDlg::OnRType()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:     // Sync the controls with the variables
 5:     UpdateData(TRUE);
 6:     // Which mode are we in?
 7:     if (m_iType == 0)    // Set the appropriate text on the button
 8:         m_ctlConnect.SetWindowText("C&onnect");
 9:     else
10:         m_ctlConnect.SetWindowText("&Listen");
11: }
```

Now, if you compile and run your application, you should be able to select one and then the other of these two radio buttons, and the text on the command button should change to reflect the part the application will play, as in Figure 20.5.

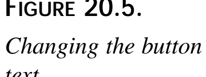
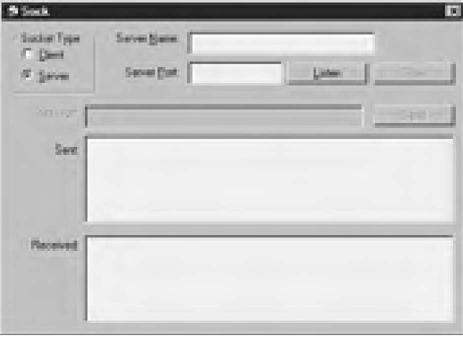


FIGURE 20.5.

Changing the button text.



Inheriting from the `CAsyncSocket` Class

So that you will be able to capture and respond to the socket events, you'll create your own descendent class from `CAsyncSocket`. This class will need its own versions of the event functions, as well as a means of passing this event to the dialog that the object will be a member of. So that you can pass each of these events to the dialog-class level, you'll add a pointer to the parent dialog class as a member variable of your socket class. You'll use this pointer to call event functions for each of the socket events that are member functions of the dialog, after checking to make sure that no errors have occurred (of course).

To create this class in your application, select Insert|New Class from the menu. In the New Class dialog, leave the class type with the default value of MFC Class. Enter a name for your class, such as `CMySocket`, and select `CAsyncSocket` from the list of available base classes. This is all that you can specify on the New Class dialog, so click the OK button to add this new class to your application.

Once you have created the socket class, add a member variable to the class to serve as a pointer to the parent dialog window. Specify the variable type as `CDialog*`, the variable name as `m_pWnd`, and the access as private. You also need to add a method to the class to set the pointer, so add a member function to your new socket class. Specify the function type as void, the declaration as `SetParent(CDialog* pWnd)`, and the access as public. Edit this new function, setting the pointer passed as a parameter to the member variable pointer, as in Listing 20.2.

LISTING 20.2. THE `CMySocket SetParent` FUNCTION.

```
1: void CMySocket::SetParent(CDialog *pWnd)
2: {
3:     // Set the member pointer
4:     m_pWnd = pWnd;
5: }
```

The only other thing that you need to do to your socket class is add the event functions, which you'll use to call similarly named functions on the dialog class. To add a function for the `OnAccept` event function, add a member function to your socket class. Specify the function type as void, the function declaration as `OnAccept(int nErrorCode)`, and the access as protected and check the virtual check box. Edit this function, adding the code in Listing 20.3.

LISTING 20.3. The `CMySocket` `OnAccept` FUNCTION.

```
1: void CMySocket::OnAccept(int nErrorCode)
2: {
3:     // Were there any errors?
4:     if (nErrorCode == 0)
5:         // No, call the dialog's OnAccept function
6:         ((CSockDlg*)m_pWnd)->OnAccept();
7: }
```

Add similar functions to your socket class for the `OnConnect`, `OnClose`, `OnReceive`, and `OnSend` functions, calling same-named functions in the dialog class, which you'll add later. After you've added all these functions, you'll need to include the header file for your application dialog in your socket class, as in line 7 of Listing 20.4.

LISTING 20.4. THE `CMySocket` INCLUDES.

```
1: // MySocket.cpp: implementation file
2: //
3:
4: #include "stdafx.h"
5: #include "Sock.h"
6: #include "MySocket.h"
7: #include "SockDlg.h"
```

Once you've added all the necessary event functions to your socket class, you'll add a variable of your socket class to the dialog class. For the server functionality, you'll need two variables in the dialog class, one to listen for connection requests and the other to be connected to the other application. Because you will need two socket objects, add two member variables to the dialog class (`CSockDlg`). Specify the type of both variables as your socket class (`CMySocket`) and the access for both as private. Name one variable `m_sListenSocket`, to be used for listening for connection requests, and the other `m_sConnectSocket`, to be used for sending messages back and forth.

Once you've added the socket variables, you'll add the initialization code for all the variables. As a default, set the application type to client, the server name as `loopback`, and the port to 4000. Along with these variables, you'll set the parent dialog pointers in your two socket objects so that they point to the dialog class. You can do this by adding the code in Listing 20.5 to the `OnInitDialog` function in the dialog class.

> **Note**
>
> The computer name `loopback` is a special name used in the TCP/IP network protocol to indicate the computer you are working on. It's an internal computer name that is resolved to the network address `127.0.0.1`. This is a computer name and address that is commonly used by applications that need to connect to other applications running on the same computer.

LISTING 20.5. THE `CSockDlg OnInitDialog` FUNCTION.

```
 1: BOOL CSockDlg::OnInitDialog()
 2: {
 3:     CDialog::OnInitDialog();
 4:
 5:     // Add "About..." menu item to system menu.
 6:
.
.
.
26:     SetIcon(m_hIcon, FALSE);        // Set small icon
27:
28:     // TODO: Add extra initialization here
29:     // Initialize the control variables
30:     m_iType = 0;
31:     m_strName = "loopback";
32:     m_iPort = 4000;
33:     // Update the controls
34:     UpdateData(FALSE);
35:     // Set the socket dialog pointers
36:     m_sConnectSocket.SetParent(this);
37:     m_sListenSocket.SetParent(this);
38:
39:     return TRUE;  // return TRUE  unless you set the focus to a
        ➥control
40: }
```


Connecting the Application

When the user clicks the Connect button, you'll disable all the top controls on the dialog. At this point, you don't want the user to think that she is able to change the settings of

the computer that she's connecting to or change how the application is listening. You'll call the `Create` function on the appropriate socket variable, depending on whether the application is running as the client or server. Finally, you'll call either the `Connect` or `Listen` function to initiate the application's side of the connection. To add this functionality to your application, open the Class Wizard and add a function to the `BN_CLICKED` event message for the Connect button (ID `IDC_BCONNECT`). Edit this function, adding the code in Listing 20.6.

LISTING 20.6. The `CSockDlg OnBconnect` FUNCTION.

```
 1: void CSockDlg::OnBconnect()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:     // Sync the variables with the controls
 5:     UpdateData(TRUE);
 6:     // Disable the connection and type controls
 7:     GetDlgItem(IDC_BCONNECT)->EnableWindow(FALSE);
 8:     GetDlgItem(IDC_ESERVNAME)->EnableWindow(FALSE);
 9:     GetDlgItem(IDC_ESERVPORT)->EnableWindow(FALSE);
10:     GetDlgItem(IDC_STATICNAME)->EnableWindow(FALSE);
11:     GetDlgItem(IDC_STATICPORT)->EnableWindow(FALSE);
12:     GetDlgItem(IDC_RCLIENT)->EnableWindow(FALSE);
13:     GetDlgItem(IDC_RSERVER)->EnableWindow(FALSE);
14:     GetDlgItem(IDC_STATICTYPE)->EnableWindow(FALSE);
15:     // Are we running as client or server?
16:     if (m_iType == 0)
17:     {
18:         // Client, create a default socket
19:         m_sConnectSocket.Create();
20:         // Open the connection to the server
21:         m_sConnectSocket.Connect(m_strName, m_iPort);
22:     }
23:     else
24:     {
25:         // Server, create a socket bound to the port specified
26:         m_sListenSocket.Create(m_iPort);
27:         // Listen for connection requests
28:         m_sListenSocket.Listen();
29:     }
30: }
```

Next, to complete the connection, you'll add the socket event function to the dialog class for the `OnAccept` and `OnConnect` event functions. These are the functions that your socket class is calling. They don't require any parameters, and they don't need to return any result code. For the `OnAccept` function, which is called for the listening socket when

another application is trying to connect to it, you'll call the socket object's `Accept` function, passing in the connection socket variable. Once you've accepted the connection, you can enable the prompt and edit box for entering and sending messages to the other application.

To add this function to your application, add a member function to the dialog class (`CSockDlg`). Specify the function type as void, the declaration as `OnAccept`, and the access as public. Edit the function, adding the code in Listing 20.7.

LISTING 20.7. THE `CSockDlg` `OnAccept` FUNCTION.

```
1: void CSockDlg::OnAccept()
2: {
3:     // Accept the connection request
4:     m_sListenSocket.Accept(m_sConnectSocket);
5:     // Enable the text and message controls
6:     GetDlgItem(IDC_EMSG)->EnableWindow(TRUE);
7:     GetDlgItem(IDC_BSEND)->EnableWindow(TRUE);
8:     GetDlgItem(IDC_STATICMSG)->EnableWindow(TRUE);
9: }
```

For the client side, there's nothing to do once the connection has been completed except enable the controls for entering and sending messages. You'll also enable the Close button so that the connection can be closed from the client side (but not the server side). To add this functionality to your application, add another member function to the dialog class (`CSockDlg`). Specify the function type as void, the function declaration as `OnConnect`, and the access as public. Edit the function, adding the code in Listing 20.8.

LISTING 20.8. THE `CSockDlg` `OnConnect` FUNCTION.

```
1: void CSockDlg::OnConnect()
2: {
3:     // Enable the text and message controls
4:     GetDlgItem(IDC_EMSG)->EnableWindow(TRUE);
5:     GetDlgItem(IDC_BSEND)->EnableWindow(TRUE);
6:     GetDlgItem(IDC_STATICMSG)->EnableWindow(TRUE);
7:     GetDlgItem(IDC_BCLOSE)->EnableWindow(TRUE);
8: }
```


If you could compile and run your application now, you could start two copies, put one into listen mode, and then connect to it with the other. Unfortunately, you probably can't even compile your application right now because your socket class is looking for several functions in your dialog class that you haven't added yet. Add three member functions to

the dialog class (`CSockDlg`). Specify all of them as void functions with public access. Specify the first function's declaration as `OnSend`, the second as `OnReceive`, and the third as `OnClose`. You should now be able to compile your application.

Once you've compiled your application, start two copies of the application, side-by-side. Specify that one of these two should be the server, and click the Listen button to put it into listen mode. Leave the other as the client and click the Connect button. You should see the connection controls disable and the message sending controls enable as the connection is made, as in Figure 20.6.

FIGURE 20.6.
Connecting the two applications.

Tip

Be sure that you have the server application listening before you try to connect it to the client application. If you try to connect to it with the client application before the server is listening for the connection, the connection will be rejected. Your application will not detect that the connection was rejected because you haven't added any error handling to detect this event.

Tip

To run these applications and get them to connect, you'll need TCP/IP running on your computer. If you have a network card in your computer, you may already have TCP/IP running. If you do not have a network card, and

you use a modem to connect to the Internet, then you will probably need to be connected to the Internet when you run and test these applications. When you connect to the Internet through a modem, your computer usually starts running TCP/IP once the connection to the Internet is made. If you do not have a network card in your computer, and you do not have any means of connecting to the Internet, or any other outside network that would allow you to run networked applications, you may not be able to run and test today's applications on your computer.

Sending and Receiving

Now that you are able to connect the two running applications, you'll need to add functionality to send and receive messages. Once the connection is established between the two applications, the user can enter text messages in the edit box in the middle of the dialog window and then click the Send button to send the message to the other application. Once the message is sent, it will be added to the list box of sent messages. To provide this functionality, when the Send button is clicked, your application needs to check whether there is a message to be sent, get the length of the message, send the message, and then add the message to the list box. To add this functionality to your application, use the Class Wizard to add a function to the clicked event of the Send (`IDC_BSEND`) button. Edit this function, adding the code in Listing 20.9.

LISTING 20.9. The `CSockDlg` `OnBsend` FUNCTION.

```
 1: void CSockDlg::OnBsend()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:     int iLen;
 5:     int iSent;
 6:
 7:     // Sync the controls with the variables
 8:     UpdateData(TRUE);
 9:     // Is there a message to be sent?
10:     if (m_strMessage != "")
11:     {
12:         // Get the length of the message
13:         iLen = m_strMessage.GetLength();
14:         // Send the message
15:         iSent = m_sConnectSocket.Send(LPCTSTR(m_strMessage), iLen);
16:         // Were we able to send it?
```

continues

LISTING 20.9. CONTINUED

```
17:         if (iSent == SOCKET_ERROR)
18:         {
19:         }
20:         else
21:         {
22:             // Add the message to the list box.
23:             m_ctlSent.AddString(m_strMessage);
24:             // Sync the variables with the controls
25:             UpdateData(FALSE);
26:         }
27:     }
28: }
```

When the `OnReceive` event function is triggered, indicating that a message has arrived, you'll retrieve the message from the socket using the `Receive` function. Once you've retrieved the message, you'll convert it into a `CString` and add it to the message-received list box. You can add this functionality by editing the `OnReceive` function of the dialog class, adding the code in Listing 20.10.

LISTING 20.10. THE `CSockDlg OnReceive` FUNCTION.

```
 1: void CSockDlg::OnReceive()
 2: {
 3:     char *pBuf = new char[1025];
 4:     int iBufSize = 1024;
 5:     int iRcvd;
 6:     CString strRecvd;
 7:
 8:     // Receive the message
 9:     iRcvd = m_sConnectSocket.Receive(pBuf, iBufSize);
10:     // Did we receive anything?
11:     if (iRcvd == SOCKET_ERROR)
12:     {
13:     }
14:     else
15:     {
16:         // Truncate the end of the message
17:         pBuf[iRcvd] = NULL;
18:         // Copy the message to a CString
19:         strRecvd = pBuf;
20:         // Add the message to the received list box
21:         m_ctlRecvd.AddString(strRecvd);
22:         // Sync the variables with the controls
23:         UpdateData(FALSE);
24:     }
25: }
```

At this point, you should be able to compile and run two copies of your application, connecting them as you did earlier. Once you've got the connection established, you can enter a message in one application and send it to the other application, as shown in Figure 20.7.

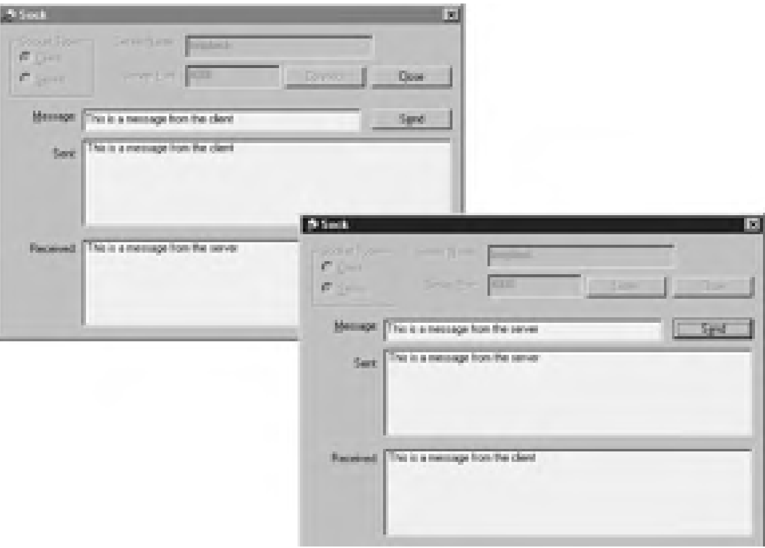


FIGURE 20.7.
Sending messages between the applications.

Ending the Connection

To close the connection between these two applications, the client application user can click the Close button to end the connection. The server application will then receive the `OnClose` socket event. The same thing needs to happen in both cases. The connected socket needs to be closed, and the message sending controls need to be disabled. On the client, the connection controls can be enabled because the client could change some of this information and open a connection to another server application. Meanwhile, the server application continues to listen on the port that it was configured to listen to. To add all this functionality to your application, edit the `OnClose` function, adding the code in Listing 20.11.

LISTING 20.11. THE `CSockDlg` `OnClose` FUNCTION.

```
1: void CSockDlg::OnClose()
2: {
```

continues

LISTING 20.11. CONTINUED

```
 3:     // Close the connected socket
 4:     m_sConnectSocket.Close();
 5:     // Disable the message sending controls
 6:     GetDlgItem(IDC_EMSG)->EnableWindow(FALSE);
 7:     GetDlgItem(IDC_BSEND)->EnableWindow(FALSE);
 8:     GetDlgItem(IDC_STATICMSG)->EnableWindow(FALSE);
 9:     GetDlgItem(IDC_BCLOSE)->EnableWindow(FALSE);
10:     // Are we running in Client mode?
11:     if (m_iType == 0)
12:     {
13:         // Yes, so enable the connection configuration controls
14:         GetDlgItem(IDC_BCONNECT)->EnableWindow(TRUE);
15:         GetDlgItem(IDC_ESERVNAME)->EnableWindow(TRUE);
16:         GetDlgItem(IDC_ESERVPORT)->EnableWindow(TRUE);
17:         GetDlgItem(IDC_STATICNAME)->EnableWindow(TRUE);
18:         GetDlgItem(IDC_STATICPORT)->EnableWindow(TRUE);
19:         GetDlgItem(IDC_RCLIENT)->EnableWindow(TRUE);
20:         GetDlgItem(IDC_RSERVER)->EnableWindow(TRUE);
21:         GetDlgItem(IDC_STATICTYPE)->EnableWindow(TRUE);
22:     }
23: }
```

Finally, for the Close button, call the `OnClose` function. To add this functionality to your application, use the Class Wizard to add a function to the clicked event for the Close button (`IDC_BCLOSE`). Edit the function to call the `OnClose` function, as in Listing 20.12.

LISTING 20.12. THE `CSockDlg` `OnBclose` FUNCTION.

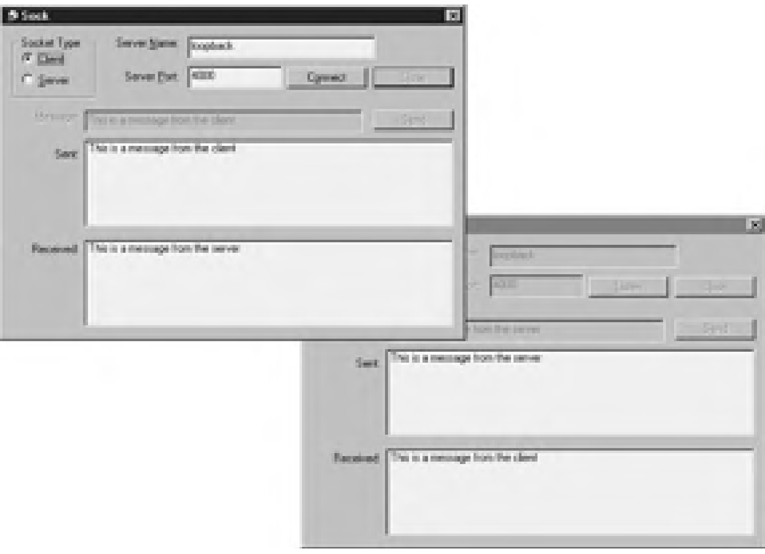
```
1: void CSockDlg::OnBclose()
2: {
3:     // TODO: Add your control notification handler code here
4:     // Call the OnClose function
5:     OnClose();
6: }
```

If you compile and run your application, you can connect the client application to the server, send some messages back and forth, and then disconnect the client by clicking the Close button. You'll see the message-sending controls disable themselves in both applications, as in Figure 20.8. You can reconnect the client to the server by clicking the Connect button again and then pass some more messages between the two, as if they had never been connected in the first place. If you start a third copy of the application,

change its port number, designate it as a server, and put it into listening mode, you can take your client back and forth between the two servers, connecting to one, closing the connection, changing the port number, and then connecting to the other.

FIGURE 20.8.

Closing the connection between the applications.



Summary

Today, you learned how you can enable your applications to communicate with others across a network or across the Internet by using the MFC Winsock classes. You took a good look at the `CAsyncSocket` class and learned how you could create your own descendent class from it that would provide your applications with event-driven network communications. You learned how to create a server application that can listen for and accept connections from other applications. You also learned how to build a client application that can connect to a server. You learned how to send and receive messages over a socket connection between two applications. Finally, you learned how to close the connection and how to detect that the connection has been closed.

Q&A

Q How do Internet applications work?

A Most Internet applications use the same functionality that you created today. The primary difference is that the applications have a script of messages that are passed back and forth. The messages consist of a command and the data that needs to

accompany that command. The server reads the command and processes the data appropriately, sending back a status code to let the client know the success or failure of the command. If you want to learn more about how Internet applications do this, several books cover this subject area in detail.

Q How does a server application handle a large number of simultaneous connections from clients?

A With a full-strength server, the connection sockets are not declared as class variables. The server instead uses some sort of dynamic allocation of sockets, in an array or link-list, to create sockets for the clients as the connection requests come in. Another approach often taken by servers is to spin off a separate thread for each connection request. This allows the application to have a single socket connection per thread, and keeping track of the sockets is much easier. In any case, server applications don't normally have a single connection socket variable.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What are the two things that a client application must know to be able to connect to a server application?
2. What `CAsyncSocket` function is used to enable a server application to detect connection efforts by client applications?
3. What `CAsyncSocket` member function is called to signal that data has arrived through a socket connection?
4. What function is called to signal that a connection has been established?
5. What function do you use to send a message through a socket connection to the application on the other end?

Exercise

The server application that you wrote can handle only a single connection at a time. If a second application tries to open a connection to it while it's got an existing connection to an application, the server application will crash. The server tries to accept the second connection into the socket that is already connected to the first client application. Add a third socket object to the application that will be used to reject additional client connections until the first client closes the connection.

WEEK 3

DAY 21

Adding Web Browsing Functionality to Your Applications

When Microsoft made the decision a few years ago to make all its applications Internet-enabled, it wasn't just talking about making Word read and write HTML pages. It wanted to make the Internet an integrated part of every application, in some way or another. Well, when it comes to development tools, making the editor double as an email client isn't really a practical integration. However, making it easy for the users of development tools to build Internet-enabled applications is a very practical feature. And this is exactly what Microsoft did.

One of the capabilities that Microsoft made available to its application development tools is using Internet Explorer as an integrated part of any application. This means that you can include Internet Explorer, and all its associated components, in your own applications. The possibilities extend far beyond

providing your users Web browsing capability; your applications can also house, and interact with, Java applets. You can provide your users with not one, but two macro languages, VBScript and JScript (Microsoft's version of JavaScript).

Today, you will learn

- How the Internet Explorer ActiveX Object Model enables you to integrate all the components into your applications.
- How the `CHtmlView` view class encapsulates most of the Internet Explorer functionality in a ready-made class.
- How to build a simple Web browser using the `CHtmlView` class and Internet Explorer.

The Internet Explorer ActiveX Model

When Microsoft came up with the idea of integrating ActiveX with its Web browser, Internet Explorer, it realized that it would need to reengineer Internet Explorer to support the use of ActiveX controls. Well, the developers looked at what they would need to do, and what was possible, and decided to make Internet Explorer a lot more than just a Web browser.

The first thing that Microsoft did was separate the Web browser from the ActiveX objects that perform all the work. As a result, it ended up with the Internet Explorer application, which is little more than an ActiveX document container, and the Internet Explorer HTML viewer control, which ran as an ActiveX document server inside the application. This meant that the Internet Explorer application could host more than just Web pages; it could also be used to host Word documents, Excel spreadsheets, PowerPoint presentations, and any other ActiveX document that had an ActiveX document server installed on the same computer, as shown in Figure 21.1.

Within the HTML viewer component, Microsoft added the capability to host other controls, including scripting engines and ActiveX controls, as shown in Figure 21.2. This gave Microsoft the flexibility to add more scripting languages to Internet Explorer as they were requested and created. This also enabled Internet Explorer to host any ActiveX controls that developers might want to create.

In designing Internet Explorer this way, Microsoft not only gave itself a lot of flexibility for future expansion of the functionality supported by Internet Explorer, but it also made the entire workings of Internet Explorer available to any developer that wants to take advantage of it and integrate Internet Explorer into his or her applications.

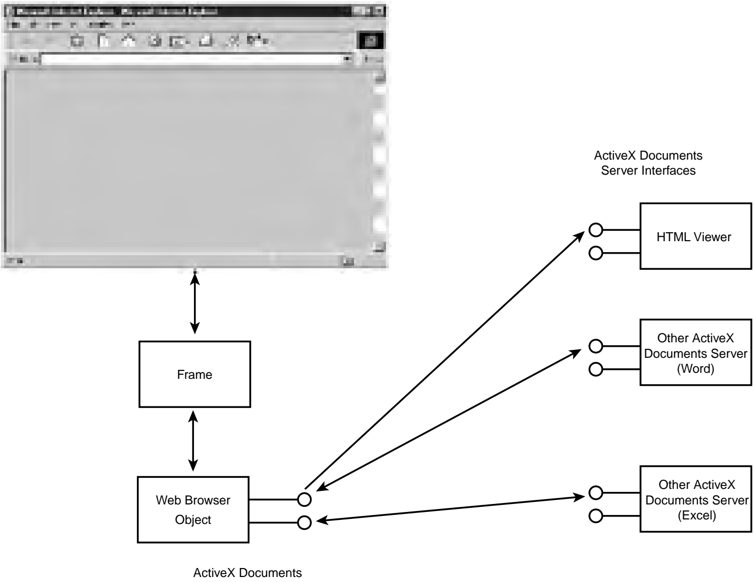


FIGURE 21.1.
The Internet Explorer ActiveX document model.

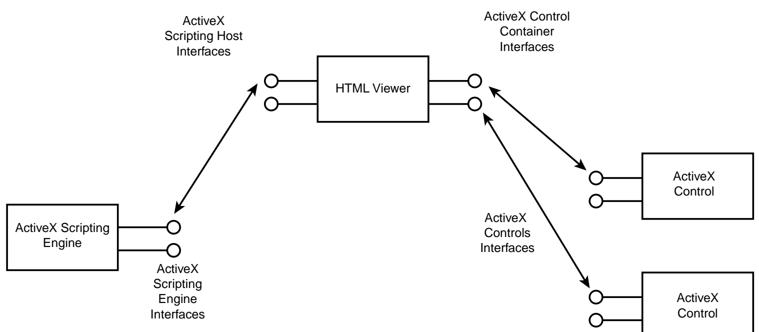


FIGURE 21.2.
The Internet Explorer HTML viewer ActiveX object model.

The `CHtmlView` Class

To make it easy to incorporate the Internet Explorer HTML viewer into Visual C++ applications, Microsoft wrapped it in the `CHtmlView` class. This class can be used as the base class for the view class in your Single Document Interface (SDI) or Multiple Document Interface (MDI) applications. You can easily create applications that have built-in Web browsing capabilities.

Navigating the Web

Several functions available in the `CHtmlView` class cover navigating the Web. There are functions for returning the browser to the starting page of the user or for taking the user

to an Internet search page. There are also functions for taking the user to the previous or next page or even to a remote Web page. All these functions are members of the `CHtmlView` class and thus are member functions of your application view class (when using the `CHtmlView` class as the base class for your view class).

The navigation functions for the `CHtmlView` class are listed in Table 21.1.

TABLE 21.1. `CHtmlView` NAVIGATION FUNCTIONS.

Function Definition	*Description*
`GoBack()`	Takes the user to the previous Web page.
`GoForward()`	Takes the user to the next Web page. (This assumes that the user has backed up from at least one Web page.)
`GoHome()`	Takes the user to the start page for Internet Explorer.
`GoSearch()`	Takes the user to an Internet search page.
`Navigate(LPCTSTR URL)`	Takes the user to the Web page specified in the URL variable.

The first four functions do not take any arguments and perform the exact same function as their toolbar equivalents in Internet Explorer. The last function does take arguments; the only required argument is the URL of the Web page to display.

Controlling the Browser

Along with the functions for navigating around the Web, you use some functions for controlling the browser. Two of these functions are `Refresh()`, which makes the HTML viewer control reload the current Web page, and `Stop()`, which halts a download in progress. As with most of the navigation functions, these functions do not take any arguments and work just like their equivalent toolbar buttons in Internet Explorer.

Getting the Browser Status

Another category of functions that are available in the `CHtmlView` class is informational in nature. You can use these functions to get information about the current state of the browser. For instance, if you want to get the current Web page in the browser, you can call `GetLocationURL()`, which returns a `CString` containing the URL. If you want to determine if the browser is busy with a download, you can call `GetBusy()`, which returns a boolean value specifying whether the browser is busy.

Many more functions are available in the `CHtmlView` class, and some of them only work on Internet Explorer itself, not on the browser control.

Building a Web-Browsing Application

For an example of how you can integrate the Internet Explorer Web browser component into your own applications, you will build a simple Web browser application. You will create an SDI application using the `CHtmlView` class as the base for your own view class. You'll add a menu with functions for the back and forward navigation options. You'll also add a dialog for getting from the user a URL that you will use to navigate the browser to the specified Web page.

Creating the Application Shell

To create a Web browser application, you can create a standard SDI or MDI application shell. The only other thing that you need to ensure is that Internet Explorer is installed on the computer where your application will run. For your development computer, this is not a problem because the Visual C++ installation probably required you to install the latest version of Internet Explorer. On any computers where you run your application, however, you might need to make sure that Internet Explorer is installed or install it yourself.

To create the shell of the application that you will build today, start a new project using the MFC AppWizard to create the application shell. Give the project a suitable name, such as `WebBrowse`, and click the OK button to start the AppWizard.

In the AppWizard, you can just as easily create an MDI Web browsing application as you can create an SDI application. For the purposes of the sample application that you are building today, go ahead and specify that the application is a Single document (SDI) application. You can accept the default settings for the rest of the AppWizard; for this example, however, choose the Internet Explorer ReBars for the toolbar appearance on the fourth AppWizard step.

Finally, on the sixth step, specify the `CHtmlView` class as the base class for your view class. This causes your application to be created using the Internet Explorer Web browser control as the main application view.

Once you finish generating the shell for your application, if you compile and run it while connected to the Internet, you'll find that you already have a working Web browser, as shown in Figure 21.3. However, you do not have the ability to specify where your browser will take you, other than clicking links in the Web pages displayed.

FIGURE 21.3.
The initial Web browsing application.

Adding Navigation Functionality

Now that you've got a working Web browser, it would be nice if you could control where it takes you. What you need to add is an edit control where the user can enter a URL. Looking at the toolbar of the running application, you notice there's a place to put this control.

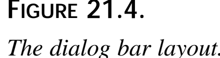
Specifying a URL

You probably noticed when you ran your application that the second toolbar had some static text telling you to lay out the dialog bar. The dialog bar is different from what you have worked with before. It is a toolbar with dialog controls on it. You even design the bar in the dialog layout designer. When you look for this dialog bar in the resource tab, you won't find it in the toolbar folder; it's in the dialogs folder.

If you open the dialogs folder and double-click the `IDR_MAINFRAME` dialog to open it in the dialog designer, you'll see that it's the second toolbar in your application. You can place edit boxes, buttons, combo boxes, and check boxes on this toolbar. You can place any control that you can use on a dialog window on this toolbar.

For your Web browser, modify the static text control already on the dialog bar and add an edit box, as shown in Figure 21.4. Specify an ID for the edit box; for this example, use the ID `IDC_EADDRESS`.

Before you open the Class Wizard to begin adding variables and event functions to the dialog bar, be aware that the dialog bar will automatically send its events to the main frame class in your application. When you open the Class Wizard, it assumes that you need to associate the dialog bar with a class and prompts you to create a new class. This association is not necessary because you can map all its events through the frame and from there feed them to the view or document classes.



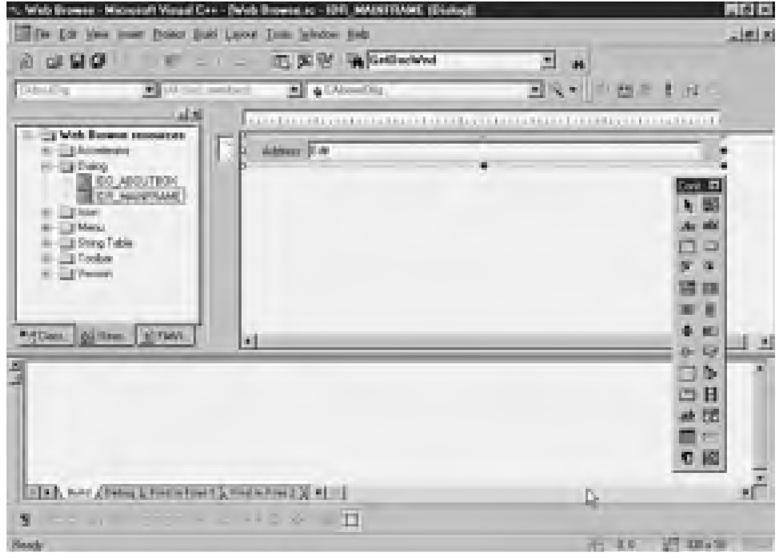


FIGURE 21.4.
The dialog bar layout.

For this example, you don't even need to use the Class Wizard to add any event handlers to the dialog bar. You need to trigger an action when the user finishes entering a URL into the edit box. The closest event available to you through the Class Wizard is the `EN_CHANGED` event, which will trigger for each letter the user types. What you need is an event that will trigger when the user presses the Enter key. Fortunately, when the user types in the edit box on the dialog bar and presses the Enter key, the `IDOK` command ID is sent to the frame class. What you can do is add a command handler in the message map to call a function on the `IDOK` command.

In your command handler, you need to get the window text from the edit box on the dialog bar. You can pass this string to the `Navigate` function in the view class, making the browser go to the page specified by the user.

To add this functionality to your application, add a new member function to the `CMainFrame` class. Specify the function type as `void`, the function declaration as `OnNewAddress`, and the access as public. Edit the new function, adding the code in Listing 21.1.

LISTING 21.1. THE `CMainFrame OnNewAddress` FUNCTION.

```
1: void CMainFrame::OnNewAddress()
2: {
3:     CString sAddress;
4:
5:     // Get the new URL
```

continues

LISTING 21.1. CONTINUED

```
6:     m_wndDlgBar.GetDlgItem(IDC_EADDRESS)->GetWindowText(sAddress);
7:     // Navigate to the new URL
8:     ((CWebBrowseView*)GetActiveView())->Navigate(sAddress);
9: }
```

In this function, line 6 got the text in the edit box using the `GetWindowText` function, placing the text into the `m_sAddress` variable. The dialog bar was declared in the `CMainFrame` class as the `m_wndDlgBar` variable, so you were able to use the `GetDlgItem` function on the dialog bar variable to get a pointer to the edit box.

In line 8, you cast the return pointer from the `GetActiveView` function as a pointer to the `CWebBrowseView` class. This allowed you to call the `Navigate` function on the view class, passing it the URL that was entered into the edit box.

Now that you are able to take the URL that the user entered and tell the browser component to go to that Web page, how do you trigger this function? You have to add the message-map entry by hand because this is one that the Class Wizard isn't able to add. In the message map, after the closing marker of the `AFX_MSG_MAP` (the section maintained by the Class Wizard), add the `ON_COMMAND` macro, specifying the `IDOK` command and your new function as the handler to be called, as in Listing 21.2. You can also add this entry before the Class Wizard section as long as it's on either side and not inside the section maintained by the Class Wizard.

LISTING 21.2. THE `CMainFrame` MESSAGE MAP.

```
1: BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
2:     //{{AFX_MSG_MAP(CMainFrame)
3:         // NOTE - the ClassWizard will add and remove mapping macros
           ➥here.
4:         //    DO NOT EDIT what you see in these blocks of generated
           ➥code !
5:     ON_WM_CREATE()
6:     //}}AFX_MSG_MAP
7:     ON_COMMAND(IDOK, OnNewAddress)
8: END_MESSAGE_MAP()
```

If you compile and run your application, you can enter a URL into the edit box on the toolbar and press the Enter key, and your application should browse to the Web page you specified, as in Figure 21.5.

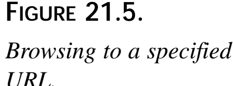


FIGURE 21.5.
Browsing to a specified URL.

Displaying the Current URL

When surfing the Web, you often follow links on Web pages that take you to other Web sites. When you do this, you wouldn't know what Web site you accessed if your browser didn't place the URL into the address box, indicating the site where you are and providing the opportunity to copy or modify the URL to find another page on the same site.

Getting the current URL from the browser is a simple matter of calling the `GetLocationURL` function and passing the result to the dialog bar. The problem is when to get the URL. It turns out that some event functions in the `CHtmlView` class can be overridden in your class. These functions will be triggered on various events that are triggered by the browser control. There are event functions for starting the navigation, beginning a download, monitoring a download's progress, and, most important for our needs, indicating a download has finished. None of these event handler functions can be added to your view class through the Class Wizard. You have to add them all.

To add the download-complete event handler to your application, add a new member function to the view class of your application. Specify the function type as `void`, the function declaration as `OnDocumentComplete(LPCTSTR lpszUrl)`, and the access as public. Edit the function, adding the code in Listing 21.3.

LISTING 21.3. THE `CWebBrowseView` `OnDocumentComplete` FUNCTION.

```
1: void CWebBrowseView::OnDocumentComplete(LPCTSTR lpszUrl)
2: {
3:     // Pass the new URL to the address bar
4:     ((CMainFrame*)GetParentFrame())->SetAddress(lpszUrl);
5: }
```

You'll notice in this function that you didn't need to call the `GetLocationURL` function after all. The URL that is downloaded is passed as an argument to this function. This allows you to pass the URL along to the frame, where you'll add another function to populate the edit box on the dialog bar with the URL.

To add the function to populate the dialog bar with the new URL, add a member function to the main frame class, `CMainFrame`. Specify the function type as `void`, the function declaration as `SetAddress(LPCTSTR lpszURL)`, and the access as public. Edit the function, adding the code in Listing 21.4.

LISTING 21.4. THE `CMainFrame SetAddress` FUNCTION.

```
1: void CMainFrame::SetAddress(LPCTSTR lpszURL)
2: {
3:     // Set the new URL in the address edit control
4:     m_wndDlgBar.GetDlgItem(IDC_EADDRESS)->SetWindowText(lpszURL);
5: }
```

In this function, you took the opposite path from the one you used to get the text from the edit box. You used the `SetWindowText` to change the text in the edit box to the URL that you are passing in. When you run your application, you should be able to see the URL address on the dialog bar change to reflect the Web page that you are viewing.

Back and Forth

Now that you can enter a URL into the dialog bar and have your application go to that Web site, and you can see the address of any Web sites that you view, it'd be nice if you could back up from where you might have gone. This is a simple matter of calling the `GoBack` and `GoForward` functions on the view class in your application. You can call these functions from menu entries, which also allows you to attach toolbar buttons to perform the same calls.

To add this functionality, open the main menu in the Menu Designer. You can delete the Edit menu from the bar, and all of the entries below it, because they are of no use in the application that you are building today. Grab the blank menu entry on the bar, and drag it to the left of the Help menu. Open the properties dialog for this menu entry and give it a caption of `&Go`. This is the menu where all navigation functions will be located.

To provide the back-and-forth functionality, you need to add two menu entries, one for the `GoBack` function and one for the `GoForward` function. Specify the properties for these two menu entries as shown in Table 21.2.

TABLE 21.2. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	`ID`	`IDM_GO_BACK`
	`Caption`	`&Back\tCtrl + B`
	`Prompt`	`Back to the previous page\nBack`
Menu Entry	`ID`	`IDM_GO_NEXT`
	`Caption`	`&Next\tCtrl + N`
	`Prompt`	`Go forward to the next page\nNext`

Once you add the menu entries, you can use the Class Wizard to add functions to the view class on both of these menu events. For the `IDM_GO_BACK` menu ID, add an event function on the `COMMAND` event message. Edit the function, adding the code in Listing 21.5.

LISTING 21.5. THE `CWebBrowseView OnGoBack` FUNCTION.

```
1: void CWebBrowseView::OnGoBack()
2: {
3:     // TODO: Add your command handler code here
4:
5:     // Go to the previous page
6:     GoBack();
7: }
```

Open the Class Wizard again, and add an event-handler function for the `IDM_GO_NEXT` object ID on the `COMMAND` event message. Edit this function with the code in Listing 21.6.

LISTING 21.6. THE `CWebBrowseView OnGoNext` FUNCTION.

```
1: void CWebBrowseView::OnGoNext()
2: {
3:     // TODO: Add your command handler code here
4:
5:     // Go to the next page
6:     GoForward();
7: }
```


Now you can run your application and use the menus to back up to the previous Web pages from wherever you surfed to and then trace your steps forward again. However, it's somewhat difficult using the menus, so what you need to do is add an accelerator for each of these menu entries.

If you open the accelerator table in the resources tree, you see a bunch of accelerators tied to menu IDs. Each of these accelerators consist of an ID and a key combination. If you right-click anywhere in the accelerator table, you see the option of adding a new accelerator to the table. Choosing this option presents you a dialog to enter the accelerator information. First, you need to specify the menu ID that the accelerator will be tied to. (As with toolbar buttons, accelerators are tied to menu entries.) Below that, you can enter the key that will trigger the accelerator, or you can select a key from the drop-down list.

On the right side of the dialog, you can select the modifiers for the key. Modifiers are the other keys that must be pressed in combination with the key that you've already specified for the accelerator to be triggered. Once you've entered all the necessary information for the accelerator, close the dialog and the information you specified is added to the table.

It's recommended that you use either the Ctrl or Alt key as one of the modifier keys on all accelerators using standard keys. If you don't use one of these two keys as part of the accelerator, your application might get confused about when the user is typing information into your application and when the user is triggering an accelerator.

To add accelerators to the back and forward menus in your application, delete the accelerator for the `ID_FILE_OPEN` menu ID because you won't use it in this application. Add a new accelerator and specify the ID as `IDM_GO_BACK` and the key as B and select the Ctrl modifier. Add a second accelerator, specifying the ID as `IDM_GO_NEXT` and the key as `N` and select the Ctrl modifier. When you run your application, you can use the Ctrl+B key combination to back up to the previous page and the Ctrl+N key combination to go forward.

To really make your application work like most available Web browsers, you would also add toolbar buttons for these two menu entries with arrows pointing to the left for back and to the right for forward.

Controlling the Browser

Often when browsing, you come across a Web page that you don't want to wait to download. You'll want to stop the transfer part-way through. Maybe you entered the wrong URL or maybe the download is taking too long. It doesn't matter why you want to stop the download; it's enough that you want to stop it. This is why the `CHtmlView` class has the `Stop` function. It cancels the download currently in progress. To add this functionali-

ty to your application, add a new menu entry to the View menu in the Menu Designer. Specify the menu entry properties in Table 21.3.

TABLE 21.3. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	`ID`	`IDM_VIEW_STOP`
	`Caption`	`Sto&p`
	`Prompt`	`Stop the current transfer\nStop`

Using the Class Wizard, add an event-handler function to the view class for this menu ID on the `COMMAND` event message. Edit the function with the code in Listing 21.7.

LISTING 21.7. THE `CWebBrowseView` `OnViewStop` FUNCTION.

```
1: void CWebBrowseView::OnViewStop()
2: {
3:     // TODO: Add your command handler code here
4:
5:     // Stop the current download
6:     Stop();
7: }
```

If you run your application, you can use this menu entry to stop any download of a Web page that you don't want to wait on. It would be more convenient if you added a toolbar button for this menu ID.

Another control function that most browsers have is the capability to reload the current Web page. This function is handy for Web pages that contain dynamic elements that change each time the page is downloaded. It's also helpful for Web pages that your browser may have in its cache so that it doesn't retrieve the newest version of the page. It's necessary to be able to force the browser to reload the page and not just display the cached version (especially if it's a Web page that you are in the process of creating). The browser component has the capability built in with the Refresh function. One call to this function means the current page is reloaded.

You can add this functionality to your application by adding another menu entry to the View menu. Specify the properties for the new menu entry using the settings in Table 21.4. You can add a separator bar between the two View menu entries that were originally there, and the two new entries, to make your menu look like the one in Figure 21.6.

TABLE 21.4. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	`ID`	`IDM_VIEW_REFRESH`
	`Caption`	`&Refresh`
	`Prompt`	`Refresh the current page\nRefresh`

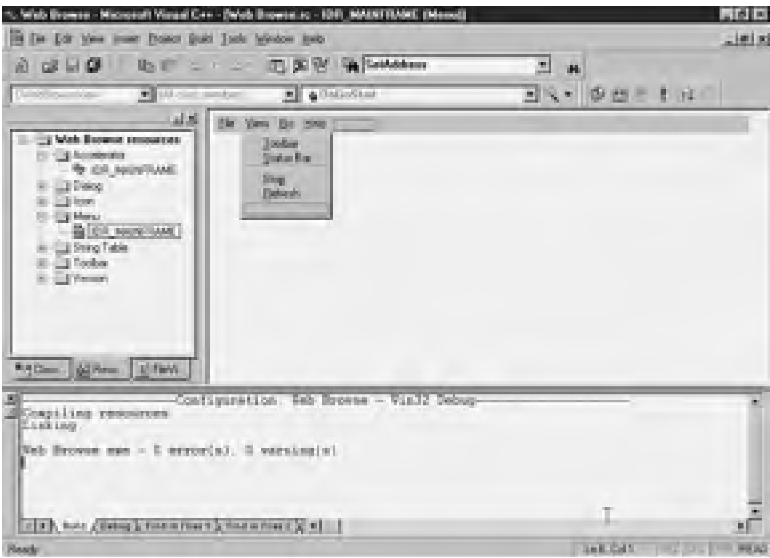


FIGURE 21.6.
The modified View menu.

Once you add the menu entry, use the Class Wizard to add an event-handler function to the view class for the `COMMAND` event message for this menu entry. Edit the function, adding the code in Listing 21.8.

LISTING 21.8. THE `CWebBrowseView OnViewRefresh` FUNCTION.

```
1: void CWebBrowseView::OnViewRefresh()
2: {
3:     // TODO: Add your command handler code here
4:
5:     // Reload the current page
6:     Refresh();
7: }
```

Now you can test this functionality by finding a Web site that returns a different page each time that you refresh the browser, such as the Random Monty Python Skit Server in

Figure 21.7. As with the rest of the menu functions that you added to this application, this one should also be added to the toolbar.

FIGURE 21.7.
The Refresh function will perform a new download.

Summary

Today you learn how Microsoft designed its Internet Explorer Web browser as a series of ActiveX components that could be used in other applications. You saw how Microsoft encapsulated the browser into the `CHtmlView` class, which can be used in SDI and MDI applications to provide Web browsing functionality to almost any application. You learned how you can use this view class to build a Web browser. You saw how you could use the dialog bar to place controls on a toolbar and how the events for these controls can be handled in the frame class for the application. Finally, you learned how to add menus to your application to call the various functions of the Web browser to provide a complete surfing experience.

Q&A

Q Why is Print Preview not included on the default menus when I choose `CHtmlView` as the base class for my view class?

A The printing for the `CHtmlView` class is performed by the browser, not the view class. You don't have print preview because the browser doesn't support it.

Q How can I get the HTML source code from the browser so that I can see or edit it?

A The `CHtmlView` class has a member function, `GetHtmlDocument`, that returns a pointer to the `IDispatch` interface of the document object in the browser. You have to use the `Invoke IDispatch` function to call the functions in the document object

to retrieve the HTML source code. For documentation on how to do this, check out the Microsoft Developer Network CD that you received with your copy of Visual C++ 6.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. The answers to the quiz questions and exercises are provided in Appendix B, "Answers."

Quiz

1. What does the `CHtmlView` class encapsulate for use in Visual C++ applications?
2. How can you get the URL for the current Web page from the `CHtmlView` class?
3. What command is triggered for the frame class when the user presses the Enter key in the edit box on the dialog bar?
4. What functions can you call to navigate the browser to the previous and the next Web pages?
5. How can you stop a download in progress?

Exercises

1. Add the `GoSearch` function to the menu and toolbar.
2. Add the `GoHome` function to the menu and toolbar.
3. Disable the Stop toolbar button and menu entry when the application is not downloading a Web page.

WEEK 3

In Review

You made it! You now have the skills to tackle most of the Windows programming tasks in Visual C++, with the exception of those that require specialized knowledge and understanding. Although there's still plenty for you to learn, you have covered the vast majority of the topics in a very short amount of time. From here you will probably want to delve into one or two more specialized areas of Windows programming—the type that an entire book can be written on—because at this point, you've got just about all the generalized skills down.

Just in case you don't have them all down, it's not a bad idea to take some time once again to come up with some of your own applications where you can apply the things you've learned. This will help pinpoint any areas that you might need to go back and review before jumping into any more advanced topics. Let's take a quick look back at what you should have learned during the past week, just to make sure.

You started off the week by learning about Microsoft's latest database application development technology, ActiveX Data Objects (ADO). You learned how ADO sits on top of the OLE DB technology, simplifying database access considerably. You learned how you can easily build a database application by using ADO as an ActiveX control and connecting it to some ADO-compatible, data-bound ActiveX controls. You also learned how you can import the DLL and dive into the code, exerting complete control over how your application deals with the database. You learned how you can read and write each of the fields in a record in the record set by converting it from and to the variant data type. You also learned how you can design your own custom record class and can

bind it to the record set object, providing you with a much more convenient way to access the individual field values in the database. If you are having problems with any of this, you might want to review Day 15 once more.

You learned three different ways of sharing the functionality modules that you develop with other programmers, without having to share your code. First you learned how to build your modules into library files that can be linked into applications by other Visual C++ developers. You saw how with this approach, if any change is made to the library module, then all the applications that it's used in have to be rebuilt. You learned how you can create these modules without making any special changes to the way you write your code. If you don't remember how you did all of this, you can go back to Day 16 to review it.

The second way that you learned to share your functionality with other programmers was to build DLLs. You learned how you can create two different types of DLLs: one that can be used only by other Visual C++ applications and one that can be used by any other application, regardless of what programming language was used to build it. You saw how you can create a DLL for use by other Visual C++ programmers without having to make any real changes to the way you design or code your modules. You also learned how you need to make dramatic changes to how your module is used and interacted with when creating DLLs that can be used by all programming languages. You learned how to provide straight function calls as an interface for use by other applications, with all necessary information to be passed in as parameters to the functions. You also learned how to build a definition file, with the names of all functions to be exported included in it. If you need any reminders of how you can do any of this, you'll want to look over Day 17 again.

The third and final way that you learned to share the functionality of your modules without sharing the code was to package it as an ActiveX control. You learned how to use the Control and Class Wizards to add in all the properties, methods, and events that you want to have in your control. You learned how to read and write the properties in your control. You saw how there are two different ways that the properties in your control can be implemented, and you learned how to determine which type is appropriate for each of your control's properties. You learned how you can raise events in the container application from your control by firing the event in your code. Along with all of this, you learned how you can use the ActiveX Control Test Container utility to test your control, calling all its methods and triggering all the events that it's capturing. You saw how you can monitor the events that your control is firing in the containing application to make sure that they are being fired as and when they should. If you need any reminders of how all this works, you can look back at Day 19 for a refresher.

An important thing that you learned was how you can enable your applications to perform more than one task at a time. This is an important piece of functionality, and more applications are requiring this capability every day. Not only did you learn how to make your applications perform multiple tasks at once, but you also learned two different ways to do so. First, you learned about the `OnIdle` function and how you can hook into this function to trigger your own functionality to be run when the application is sitting idle. You also learned about the shortcomings of using this approach to adding a second task to your application and how it can prevent your application from responding to the user. You need to slice the background task into little pieces that can be done quickly, which requires you to develop some elaborate way of keeping track of where the task is and where it needs to pick back up when the application is idle again.

The second way that you learned to give your applications a second or third task to do is by spinning off separate threads, which run completely independent of the main user-interface thread. You learned how to create a callback function that controls the top level of execution for the threads and how you can start and stop the thread as necessary. You also saw how these independent threads are completely independent from the rest of the application and how they'll continue to run, even when the rest of the application is also busy. If you feel the need to look at all this a second time, you might want to read Day 18 again.

Another area of growing importance that you learned about was how to build Internet applications using the Winsock interface classes. You learned how you can build one application that connects to another over a network and sends messages back and forth. You learned that, just like with a telephone, for one application to connect to another, the second application has to be listening for the connection. You saw how easy it is to send messages and to be notified when a message has arrived after the connection between the two applications has been made. If you need to review some of this, you might want to look back at Day 20.

Finally, you learned how you can incorporate the Microsoft Internet Explorer Web browser into your application without any effort whatsoever. You learned how you can control the browser by specifying the URL that it should load and display for the user and how you can display informational messages to the user to show what the browser is doing and when it's busy. If you need to look back at this to refresh your memory, you can go back to Day 21.

That's it. You're done. You've covered a lot of ground and learned some advanced topics, especially over this last week. Now it's time to put this book down and get busy programming, building your own applications using what you've learned. Good luck. If you find that you need a little help or advice, a good place to turn is the Microsoft newsgroups on the Internet. They are full of people who are both knowledgeable and helpful.

APPENDIX A

C++ Review

The appendix is designed to provide you with a quick review of the fundamentals of the C++ programming language. After reading this appendix, you will have a thorough understanding of the various aspects of C++ and its syntax.

Creating Your First Application

Your first example is a simple program that displays "Hello World" on the screen. For this, you create a workspace and the C++ file required for the program. The procedure for writing a C++ program using Visual C++ is simple and easy. Follow these steps:

1. From the main menu, select Visual C++.
2. Select File|New from the Visual C++ toolbar.

   Make sure the Projects tab is selected (see Figure A.1).
3. Select Win32 Console Application from the options on the left.

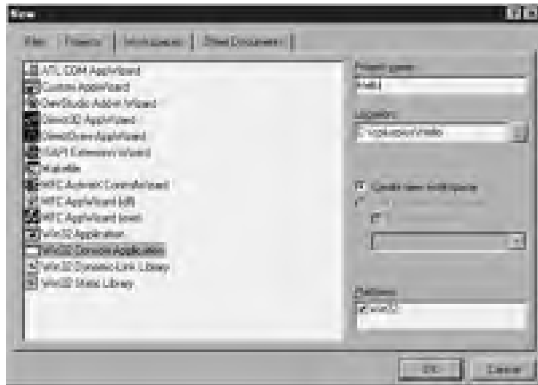


FIGURE A.1.

Setting up the Hello workspace.

4. Type **`Hello`** on the right side under Project Name.
5. Select OK.

Visual C++ creates the workspace of your application. Visual C++ actually creates a directory `Hello`, which enables you to store all files related to a particular project in one area. You will begin adding the files you require for this project:

1. Once again, select File|New from the toolbar.
2. Select the Files tab if it is not already selected.
3. Highlight C++ Source File.
4. Check the Add to Project box on the right side.
5. In the File Name edit box, type **`Helloworld`** (see Figure A.2).
6. Click OK.

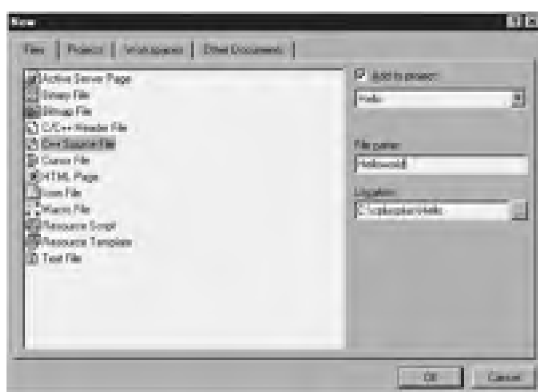


FIGURE A.2.

Setting up the Helloworld project.

The `Helloworld.cpp` file is where you add the C++ source code. All C++ source code files have a `.cpp` extension. Later, I will cover other file types.

You create all the tutorial examples in this section in a similar way. The only difference is that the names of the workspaces and the files are different.

Helloworld.cpp

The Helloworld program displays `HELLO WORLD` on the screen. Listing A.1 contains the code. Type the code exactly as shown in the `Helloworld.cpp` window. Do not type the line numbers; they are for reference only. C++ is case sensitive, so `main` is not the same as `MAIN`, which is not the same as `Main`.

LISTING A.1. `Helloworld.cpp`.

```
 1: // Workspace Name:   Hello
 2: // Program Name:  Helloworld.cpp
 3:
 4: # include <iostream.h>
 5:
 6: int main()
 7:
 8: {
 9: cout<< "HELLO WORLD \n";
10: return 0;
11: }
```

To run the program, follow these steps:

1. Select File|Save to save your work.
2. Select Build|Set Active Configuration (see Figure A.3).
3. Highlight Hello - Win32 Debug and click OK (see Figure A.4).
4. Select Build|Build `Hello.exe`.

Visual C++ compiles and links the program to create an executable file. The configuration window indicates the success or failure of the compilation. A successful compilation returns

```
Hello.exe - 0 error(s), 0 warning(s)
```

If you encounter any errors, verify that all the lines of the program were typed exactly as shown.

To execute the Helloworld program, select Build|Execute `Hello.exe`.

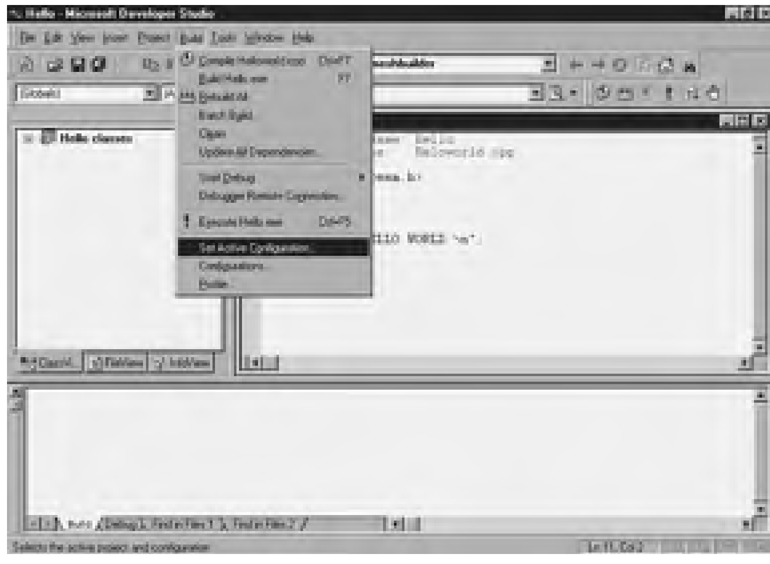


FIGURE A.3.
Setting the active configuration.

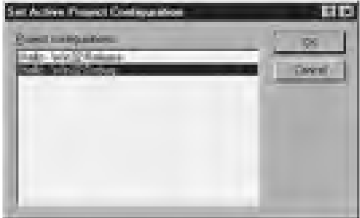


FIGURE A.4.
Selecting Win32 Debug.

The program executes by opening an MS-DOS shell and displaying the text `HELLO WORLD` (see Figure A.5).

FIGURE A.5.
`HELLO WORLD` *display.*

A

Components of `Helloworld.cpp`

The first two lines of the program are comment lines:

```
// Workspace Name:   Hello
// Program Name:  Helloworld.cpp
```

The double slash command (//) tells the compiler to ignore everything after the slash. It is good programming practice to comment your work because it makes the program easier to read, especially for someone who did not write it. Comments become important when you are working on a complex program for months. When you want to make changes, comments assist you in recollecting your thoughts from more than a month ago.

The third line begins with the pound symbol (#):

```
# include <iostream.h>
```

This is a directive to the preprocessor to search for the filename that follows (`iostream.h`) and include it. The angled brackets (< >) cause the preprocessor to search for the file in the default directories. The `iostream.h` file contains definitions for the insertion (<<) and extraction (>>) operators. This directive is required to process the `cout` statement defined on line 9 in the program. The `iostream.h` file is a precompiled header provided with your compiler. You may experiment with the Helloworld program by commenting out the include line. To do this, insert the backslash (//) before the pound sign (#). When you compile and execute this program, you get an error:

```
Compiling...
Helloworld.cpp
C:\cplusplus\Hello\Helloworld.cpp(9) : error C2065:
➥ 'cout' : undeclared identifier
C:\cplusplus\Hello\Helloworld.cpp(9) : error C2297: '<<' : bad right
➥operand
Error executing cl.exe.

Hello.exe - 2 error(s), 0 warning(s)
```

Without the `iostream.h` file, the program does not recognize the `cout` command or the insertion operator (<<).

The next line of code, line 6, is actually where program execution begins. This is the entry point of your code:

```
int main()
```

This line tells the compiler to process a function named `main`. Every C++ program is a collection of functions. You will cover functions in greater detail later in this appendix. For now, you define a function as the entry point for a block of code with a given name. The empty parentheses indicate that the function does not pass any parameters. Passing

parameters by functions is described in the section "Functions and Variables," later in this chapter.

Every C++ program must have the function `main()`. It is the entry point to begin program execution. If a function returns a value, its name must be preceded by the type of value it will return; in this case, `main()` returns a value of type `int`.

The block of code defined by any function should be enclosed in curly brackets (`{ }`):

```
{
cout<< "HELLO WORLD \n";
return 0;
}
```

All code within these brackets belongs to the named function—in this case, `main()`.

The next line executes the `cout` object. It is followed by the redirection operator `<<`, which passes the information to be displayed. The text to be displayed is enclosed in quotes. This is followed by the newline operator (`\n`). The redirection or insertion operator (<<) tells the code that whatever follows is to be inserted to `cout`.

Line 9 ends with a semicolon. All statements in C++ must end with a semicolon.

Line 10 of the code has a `return` statement. Programmers often use `return` statements either to return certain values or to return errors. Also remember that on line 7 when you defined the `main()` function, you defined its return type to be an integer (`int`). You may rerun this code by deleting the return statement on line 10, in which case line 7 would have to be modified as follows:

```
void main()
```

It is good programming practice to include return codes for complex programs. They will help you identify and track down bugs in your program.

Functions and Variables

The Helloworld program consists of only one function, `main()`. A functional C++ program typically consists of more than a single function. To use a function, you must first declare it. A function declaration is also called its prototype. A prototype is a concise representation of the entire function. When you prototype a function, you are actually

writing a statement, and as mentioned before, all statements in C++ should end with semicolons. A function prototype consists of a return type, name, and parameter list. The return type in the `main()` function is `int`, the name is `main`, and the parameter list is `()`, `null`.

A function must have a prototype and a definition. The prototype and the definition of a function must agree in terms of return type, name, and parameter list. The only difference is that the prototype is a statement and must end with a semicolon. Listing A.2 illustrates this point with a simple program to calculate the area of a triangle.

LISTING A.2. `Area.cpp`.

```
 1: // Workspace:  Triangle
 2: // Program name:  Area.cpp
 3: // The area of a triangle is half its base times height
 4: // Area of triangle = (Base length of triangle * Height of triangle)/2
 5:
 6: #include <iostream.h>       // Precompiled header
 7:
 8: double  base,height,area;    // Declaring the variables
 9: double Area(double,double);   // Function Prototype/declaration
10:
11: int  main()
12: {
13: cout << "Enter Height of Triangle: ";  // Enter a number
14: cin >> height;              // Store the input in variable
15: cout << "Enter Base of Triangle: ";   // Enter a number
16: cin >> base;              // Store the input in variable
17:
18: area = Area(base,height);        // Store the result from the Area
    ➥function
19:                                  // in the variable area
20: cout << "The Area of the Triangle is: "<< area << endl ; // Output the
    ➥area
21:
22: return 0;
23: }
24:
25: double  Area (double base, double height)    // Function definition
26: {
27: area = (0.5*base*height);
28: return area;
29: }
```

This program declares three variables, `base`, `height`, and `area`, on line 8. Variables store values that are used by the program. The type of a variable specifies the values to be stored in the variable. Table A.1 shows the various types supported by C++.

TABLE A.1. VARIABLE DATA TYPES.

Variable Data Type	*Values*
`unsigned short int`	0 to 65,535
`short int`	–32,768 to 32,767
`unsigned long int`	0 to 4,294,967,925
`long int`	–2,147,483,648 to 2,147,483,647
`int`	–2,147,483,648 to 2,147,483,647 (32 bit)
`unsigned int`	0 to 4,294,967,295 (32 bit)
`char`	256 character values
`float`	1.2e–38 to 3.4e38
`double`	2.2e–308 to 1.8e308

To define a variable, you first define its type, followed by the name. You may also assign values to variables by using the assignment (=) operator, as in these two examples:

```
double base = 5;

unsigned long int base =5;
```

In C++, you may also define your own type definition. You do this by using the keyword `typedef`, followed by the existing type and name:

```
typedef unsigned long int ULONG;

ULONG base =5;
```

Defining your own type does save you the trouble of typing the entire declaration.

The next line of the code, line 9, defines the prototype of your function:

```
double Area (double,double);
```

This function has a type `double`, a name `Area`, and a parameter list of two variables of type `double`. When you define the prototype, it is not necessary to define the parameters, but it is a good practice to do so. This program takes two inputs from the user, namely `base` and `height` of the triangle, and calculates the `area` of the triangle. The `base`, `height`, and `area` are all variables. The `Helloworld.cpp` example used the insertion (<<) operator. In this example, you use the extraction (>>) operator. The program queries the user to enter a value for the height of the triangle on line 13. When the user enters a value for `height`, the data from the screen is extracted and placed into the variable `height`. The process is repeated for the base of the triangle on lines 15 and 16. After

accepting the input from the user, the function `main()` passes execution to the function `Area(base,height)` along with the parameter values for `base` and `height`. When `main()` passes the execution to the function `Area(base, height)`, it expects a value of type `double` in return from the function. The calculation of the area of the triangle is conducted on line 27:

```
area = (0.5*base*height);
```

`Area` is the name of a function, and `area` is a variable name. Because C++ is case sensitive, it clearly distinguishes these two names.

This statement uses the standard operators, the assignment operator (=), and the multiplication operator (`*`). The assignment operator assigns the result of `(0.5*base*height)` to the variable area. The multiplication operator (`*`) calculates the resulting values of `(0.5*base*height)`. The assignment operator (=) has an evaluation order from right-to-left. Hence, the multiplication is carried out prior to assigning the values to `area`. The five basic mathematical operators are addition (+), subtraction (-), multiplication (`*`), division (`/`), and modulus (`%`).

Line 28 of the `Area` function returns the value of the variable `area` to the `main()` function. At this point, the control of the program is returned to line 18 of the `main()` function. The remainder of the program displays the result of `area` to the screen.

The `if` Statement, Operators, and Polymorphism

While programming large complex programs, it is often necessary to query the user and provide direction to the program based on his input. This is accomplished by using the `if` statement. The next example demonstrates the application of an `if` statement. The format of the `if` statement is

```
if (this expression)
   do this;
```

The `if` statement is often used in conjunction with relational operators. Another format of the `if` statement is

```
if (this expression)
   do this;
else
   do this;
```

Because `if` statements often use relational operators, let's review relational operators. Relational operators are used to determine if two expressions or numbers are equal. If

the two expressions or numbers are not equal, the statement will evaluate to either `0` or `false`. Table A.2 lists the six relational operators defined in C++.

TABLE A.2. RELATIONAL OPERATORS.

Operator	*Name*
==	Comparative
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

C++ also has logical operators. The advantage of logical operators is the ability to compare two individual expressions and conclude whether they are true or false. Table A.3 lists the three logical operators.

TABLE A.3. LOGICAL OPERATORS.

Symbol	*Operator*
&&	AND
¦¦	OR
!	NOT

An important and powerful feature of C++ is function overloading, or polymorphism. *Polymorphism* is the ability to have more than one function with the same name that differ in their parameter lists. The next example is an extension of the previous triangle code. In this program, you will calculate the area of a triangle and a circle. You will be asked whether you want to calculate the area of a triangle or a circle. Depending upon your response, `1` for triangle and `2` for circle, the program collects your input and calculates the area. In Listing A.3, the `Area` function is overloaded. The same function name is used to calculate the area of the triangle or the circle. The functions differ only in their parameter lists.

LISTING A.3. `Overload.ccp.`

```
1: // Workspace Name:  Overload
2: // Program Name:  Overload.cpp
3:
4: # include <iostream.h>
5:
```

A

```
 6: double  base,height,radius;        // Global variables
 7: double  Area_of_triangle,Area_of_circle;  // Global variables
 8: int  choice;                // Global variable
 9:
10: double  Area (double,double);       // Function prototype
11: double  Area (double);           // Function prototype
12:
13: const double  pi = 3.14;          // Constant variable
14:
15: void main()                 // main function
16:
17: {
18:   cout << "To find the area of a Triangle, input 1 \n";
19:   cout << "To find the area of a Circle, input 2 \n";
20:   cin >> choice;
21:
22: if (choice == 1)
23:
24: {
25:   cout << "Enter the base of the triangle: ";
26:   cin >> base;
27:   cout << "Enter the height of the triangle: ";
28:   cin >> height;
29:
30: Area_of_triangle = Area(base,height);
31:
32:   cout << "The Area of the Triangle is: "<<Area_of_triangle<<endl;
33: }
34:
35: if (choice == 2)
36:
37: {
38:   cout << "Enter radius of the Circle: ";
39:   cin >> radius;
40:   Area_of_circle = Area(radius);
41:   cout << "The area of the Circle is: "<<Area_of_circle<<endl;
42: }
43:
44: if (choice != 1 && choice != 2)
45:
46: {
47:   cout << "Sorry! You must enter either 1 or 2 \n";
48: }
49: }
50:
51: double Area (double base, double height)
52: {
53:   return (0.5*base*height)
54: }
```

continues

LISTING A.3. CONTINUED

```
55:
56: double Area(double radius)
57: {
58:   return (pi*radius*radius);
59: }
```

Global and Local Variables

In all of the preceding examples, the variables have been declared at the beginning of the program, prior to defining the `main()` function. Declaring variables in this fashion is more akin to C programs than C++. They are global variables and can be accessed by all the functions. However, you may also define local variables that have a scope only in a particular function. Local variables can have the same names as the global variables, but they do not change the global variables. Local variables refer only to the function in which they are defined. This difference can be confusing and lead to erratic results.

The program in Listing A.4 clearly shows the difference between global and local variables. You will calculate the area of a circle using global variables and local variables.

LISTING A.4. `Global.cpp.`

```
 1: // Workspace:  Variable
 2: // Program name:  Global.cpp
 3:
 4: #include <iostream.h>
 5:
 6: double  area;
 7: double Area (double);
 8: const double pi = 3.14;
 9: double  radius = 5;
10:
11: int main()
12:
13: {
14: cout<<"This Program Calculates The Area Of A Circle \n";
15: area = Area (radius);
16: cout << "The Area of the Circle is: "<<area<<endl;
17: cout << "The Radius In the Main() Function is: "<<radius<<endl;
18: return 0;
19: }
20:
21: double  Area (double radius)
22: {
23: area = (pi*radius*radius);
24: cout<<"The Radius In the Area() Function is: "<<radius<<endl;
```

```
25: return area;
26: }
```

The variable `radius` is accessible in the `main()` function and also the `Area()` function, and it is the same. The result of executing this program is shown in Figure A.6.

FIGURE A.6.
`Global.cpp`—*using a global variable.*

As the program executes, it shows the value of the variable `radius` in the different functions. You will now modify the global variable to be a local variable. Add an additional line to the `Area` function defining a local variable:

```
double radius = 2;
```

Compile and execute this program. The results are shown in Figure A.7.

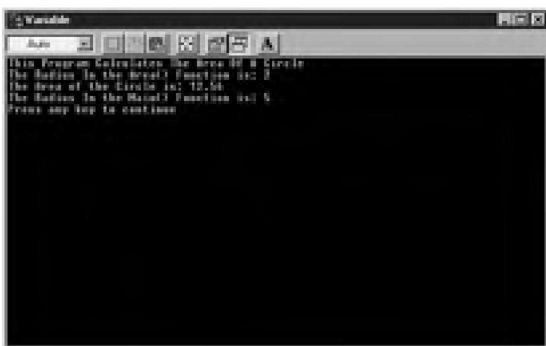


FIGURE A.7.
`Global.cpp`—*global and local variables.*

You will notice that the value of the variable `radius` remains unchanged in the `main()` function and changes locally in the `Area()` function. The area of the circle is calculated based on the value of the local variable, whereas at the same time, the value of the global variable is not changed but is hidden from the function `Area()`.

It is always advisable to differentiate your global and local variables by prefixing them with a `g` for global and `l` for local.

Pointers

Pointers are one of the most important features of C++, and they are always confusing to new programmers of C++. Pointers work by providing access to the original data directly, which increases efficiency. Pointers primarily work with two operators, the indirection operator (`*`) and the address-of operator (`&`). It is common practice to add a `p` to the beginning of a pointer variable's name to distinguish it from other variables. A pointer is just another variable, but the difference is it holds a memory address. You declare a pointer by putting an asterisk (`*`) in front of the pointer name. To access the address of the variable, you put the `&` operator in front of the variable name.

To understand pointers, you need a brief overview of how variables are stored. You covered different variable types in Table A.1. Table A.4 shows the size of the variable types.

TABLE A.4. VARIABLE TYPE SIZES.

Variable Type	*Size in Bytes*
`unsigned short int`	2 bytes
`short int`	2 bytes
`unsigned long int`	4 bytes
`long int`	4 bytes
`int`	4 bytes (32 bit)
`unsigned int`	2 bytes(32 bit)
`char`	1 byte
`float`	4 bytes
`double`	8 bytes

In the program `address.cpp` in Listing A.5, the two variables `base` and `radius` each occupy 8 and 4 bytes. Assume that your computer memory has a certain space to store these variables, they are sequentially numbered from 1 through 12, and each space is 1 byte. When you declare the variable `base` of type `double`, it occupies 8 bytes. Assume these 8 bytes reside at locations beginning from 1 through 8. You also declared another

variable `radius` of type `int`, which occupies 4 bytes and its location is byte 9 through byte 12. The location of each of these variables is termed as its address. Hence, the variable `base` has an address beginning at address 1 and ending at address 8. Similarly, the variable `radius` has an address beginning at address 9 and ending at address 12. When you use the address-of operator (`&`) on a variable, this is the address returned. The variable `base` has an address from 1 through 8, but the address-of operator returns its address as 1. Internally, the system already knows that the total addresses occupied are 8 because you defined its type as `double`.

Note

> The byte size shown in Table A.4 is not fixed. It can be different depending on your compiler and the hardware on which it runs. To determine the size of the variable for your individual compiler and hardware settings, use the `sizeof()` function as implemented in Listing A.5 on lines 13 and 16.

The program in Listing A.5 shows how to access the memory address of variables.

LISTING A.5. `Address.cpp`.

```
 1: // Workspace:  Pointers
 2: // Program name:  Address.cpp
 3:
 4: #include <iostream.h>
 5:
 6: double base = 5.0;
 7: int radius = 2;
 8:
 9: void main()
10: {
11: cout<<"The VALUE of base is: "<<base<<endl;
12: cout<<"The ADDRESS of base is: "<<&base<<endl;
13: cout<<"The SIZE of double base is: "<<sizeof(double)<< "bytes \n";
14: cout<<"The VALUE of radius is: "<<radius<<endl;
15: cout<<"The ADDRESS of radius is: "<<&radius<<endl;
16: cout<<"The SIZE of integer radius is: "<<sizeof(int)<<" bytes \n";
17: }
```

The address of the variables is accessed directly on lines 12 and 15 by using the address-of operator (`&`).The addresses of the variables `base` and `radius` are shown in Figure A.8. The addresses of the variables depend on your system, so they might not be the same.

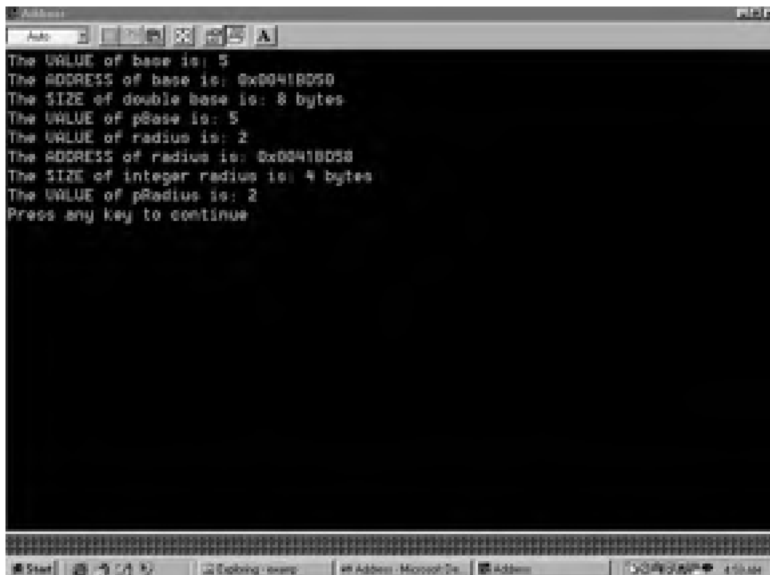


FIGURE A.8.
Using the address-of operator.

The indirection operator (`*`) operates by providing access to the value stored in the address of the variable. When a pointer is declared for a specific variable type (such as `int`), it should not be used with any other type unless it is recast to a new type. You should remember that a pointer is a variable, and like all other variables, it should be declared and initialized. A pointer that is not initialized could be dangerous. The program in Listing A.5 is modified to access the values of the variables `radius` and `base`. The modified program is provided in Listing A.6.

LISTING A.6. `Address.cpp.`

```
 1: // Workspace:  Pointers
 2: // Program name:  Address.cpp
 3:
 4: #include <iostream.h>
 5:
 6: double base =5.0;
 7: int radius =2;
 8:
 9: double *pBase =0;           // Initialize the pointer variable
10: int *pRadius =0;            // Initialize the pointer variable
11:
12: void main()
13: {
14: pBase = &base;                   // Assign the address of base
```

A

```
15: pRadius = &radius;               // Assign the address of radius
16: cout<<"The VALUE of base is: "<<base<<endl;    // Output value of base
17: cout<<"The ADDRESS of base is: "<<&base<<endl; // Output address of
       ➥base
18: cout<<"The SIZE of double base is: "<<sizeof(double)<< "bytes \n
19: cout<<"The VALUE of pBase is: "<<*pBase<<endl;
       ➥ // Output redirected value of base
20:
21: cout<<"The VALUE of radius is: "<<radius<<endl;
       ➥// Output value of radius
22: cout<<"The ADDRESS of radius is: "<<&radius<<endl;
       ➥// Output address of base
23: cout<<"The SIZE of integer radius is: "<<sizeof(int)<<" bytes \n";
24: cout<<"The VALUE of pRadius is: "<<*pRadius<<endl;
       ➥// Output redirected value of radius
25:
26: }
```

References

An important feature in C++ that is used often with function parameters is *references*. Reference is simply a synonym for variable. Until now, you have passed parameters in functions by value. You will learn how to pass parameters by reference. You create a reference variable by specifying its type and preceding the name with the reference operator (`&`). If you have a variable `float radius`, you create a reference with

```
void functionname (float &rfradius);
```

You can give the reference variable any name you want; in the following example, the reference variable names have an `rf` prefix. The advantage of a reference is that you can pass it as a parameter, like any other variable. However, unlike regular parameters, changes made to the reference's value while in a function are stored in the original variable. The example in Listing A.7 shows how the reference changes the value of the variable in the `main()` function.

LISTING A.7. `Refer.cpp`.

```
 1: // Workspace:  Reference
 2: // Program name:  Refer.cpp
 3:
 4: #include <iostream.h>
 5:
 6: void squareit (float &num);
 7: int main()
 8:
```

continues

LISTING A.7. CONTINUED

```
 9: {
10: float num=5.0;
11:
12: cout<<"In Main: before squaring number: "<<num*num<<"\n";
13:
14: squareit (num);
15: cout<<"In Main: after squaring number: "<<num*num<<"\n";
16: return 0;
17:
18: }
19:
20: void  squareit (float &rfnum)
21: {
22:
23: cout<<"In Squareit: before squaring number: "<<rfnum*rfnum<<"\n";
24:
25: rfnum = rfnum+5;
26:
27: cout<<"In Squareit: after squaring number: "<<rfnum*rfnum<<"\n";
28:
29: }
```

You define a function `squareit` on line 6, and its parameters are references. This is the function prototype. On line 10, the variable `num` is given a value of `5`. The square of the number is displayed to the screen on line 15. On line 14, you call the `squareit` function.

You pass the variable `num` and not its address.

Only when execution jumps to line 20 from line 14 are the variables identified as references. On line 27, the references are squared and displayed. They should be the same as the variables because they are just like aliases for the variables. On line 25, you add 5 to the reference, which in turn changes the variable `num`. The incremented value is squared and displayed to the screen. Execution returns to `main()` on line 15, where the display confirms the variable was changed. The output for this program is shown in Figure A.9.

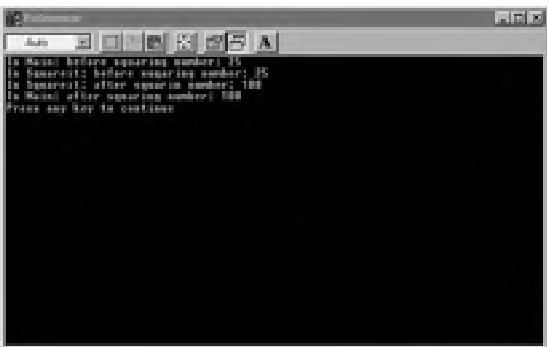


FIGURE A.9.
Passing parameters by reference.

Classes

In the previous sections, you used data types (`int`, `float`, and so on) that are inherently built into C++. In large complex programs, it is easier to define your own type, which could be a combination of the inherent types. Classes were added to C++ primarily for this purpose—to enable the programmer to be able to define custom data types and methods. The concept of classes in C++ evolved due to certain limitations of the concept of structures in C. To thoroughly understand classes, you have to step back into C and understand structures first.

A structure in C/C++ is a way of representing your own custom data. When you defined variables, you first defined their data types, followed by their names:

```
int radius;
```

To define your own data types, you use the keyword `struct`. The syntax for declaring a structure is

```
struct [structure_name]
  {
  data_members
  }
```

The *`data_members`* of a structure are variables and functions. When functions are associated with classes, they are more appropriately referred to as methods. From now on, you use the term function for program code that is not a part of a structure or class. A reference to methods indicates that the function is associated with a class structure. To understand how structures are used, review the example in Listing A.8.

LISTING A.8. `Struct.cpp`.

```
 1: // Workspace Name: Class1
 2: // Program Name:  Struct.cpp
 3: #include <iostream.h>
 4:
 5: struct farm_house
 6: {
 7: int pig_values;
 8: };
 9:
10: int main()
11: {
12: farm_house pig1, pig2, pig3;
13:
14: pig1.pig_values = 12;
15: pig2.pig_values = 13;
16: pig3.pig_values = 14;
17:
18: cout << "The value of pig1 is " << pig1.pig_values<< "\n";
19: cout << "The value of pig2 is " << pig2.pig_values << "\n";
20: cout << "The value of pig3 is " << pig3.pig_values << "\n";
21:
22: return 0;
23: }.
```

On line 5, the `struct` keyword is followed by the name of the structure. The actual definition of the structure is enclosed in the curly brackets. This particular structure defines a data member of type `int` and name `pig_value`. If you remember, I mentioned earlier that when you define a structure, you basically define a custom-made data type. All data types end with a semicolon, so the structure should also end with a semicolon. On line 12, you define three instances of the same type of `farm_house`, each of which contains a single `int` type variable.

If you strictly use C, then to define instances on line 12, you must use the keyword `struct`:

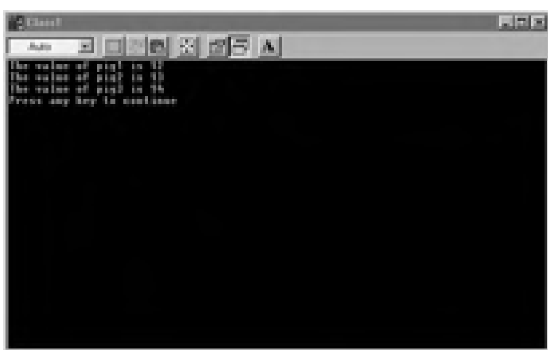
```
struct farm_house pig1, pig2, pig3;
```

This is no longer required in C++.

On lines 14 through 16, you assign values to the member variables of each structure. The structure member operator (.), also called the dot operator, is used to access member variables of the structure. On lines 18 through 20, the assigned values are output to the screen. Figure A.10 shows the output from this program.

FIGURE A.10.

Structure output.



The most important concept of object-oriented programming is encapsulation. Encapsulation can involve one or more classes. Encapsulation promotes safeguards and data hiding. The `struct.cpp` program had no encapsulation or classes. What do encapsulation and classes mean in object-oriented programming?

Let's start with describing the syntax and components of a class:

```
class class_name
{
public:
class_name_constructor;
~class_name_destructor;
class_method_prototypes();
class_member_variables;
private:
class_method_prototypes();
class_member_variables;
};
```

The words in bold are keywords. You declare a class by using the `class` keyword. This is followed by the name of the class. The data and methods of a class are enclosed in curly brackets ({ }). The methods of a class are function prototypes. They determine the behavior of the objects of your class. The member variables are the variables in your class. Classes have constructors and destructors. The methods and variables can be classified as either `public` or `private`.

You will now re-create the previous example of `Struct.cpp` in Listing A.8, employing the class and encapsulation methodology. The output from this program in Listing A.9 is identical to the previous example, `Struct.cpp`.

LISTING A.9. `Clasfarm.cpp.`

```
 1: // Workspace:  Class2
 2: // Program Name:  Clasfarm.cpp
 3: #include <iostream.h>
 4:
 5: class farm_house
 6: {
 7: int pig_values;
 8: public:
 9:  void set(int input);
10:  int get(void);
11: };
12:
13: void farm_house::set(int input)
14: {
15:   pig_values = input;
16: }
17:
18: int farm_house::get(void)
19: {
20:  return pig_values;
21: }
22:
23: int main()
24: {
25: farm_house pig1, pig2, pig3;
26:
27:
28:  pig1.set(12);
29:  pig2.set(13);
30:  pig3.set(14);
31:
32:  cout << "The value of pig1 is " << pig1.get() << "\n";
33:  cout << "The value of pig2 is " << pig2.get() << "\n";
34:  cout << "The value of pig3 is " << pig3.get() << "\n";
35:
36:  return 0;
37:
38: }
```

Compare the `struct` declaration of the `Struct.cpp` program in Listing A.8 (lines 5 through 7) to the class declaration of the `Clasfarm.cpp` program in Listing A.9 (lines 5

through 11). The difference is in the private and public portions of their declarations. In the `struct` declaration, everything is public, whereas in the class declaration, you begin with a private section. All data and methods at the beginning of a class are private. This means the member variable

```
int pig_values;
```

is private and hidden to methods outside the class. This means that the variable `pig_values` is not accessible inside `main()`. In other words, this member variable is hidden. This member variable is accessible to the methods of its class, mainly

```
void set (int input);
int get(void);
```

These methods are defined to be public. Because they are public, these methods can be accessed by any objects of this class. On line 25, you defined `pig1`, `pig2`, and `pig3` to be instances or objects of the class. What? I am sure you are wondering why `pig1` is an object.

You defined on line 5 a class `farm_house`. Remember when you declare a class, all you are doing is declaring a new type. When you declare a variable, you declare its type and then the variable name, as shown here:

```
long somevariable, anotherone, onemore;
```

Similarly, to define an object of a class, you declare the type, which in this case is `farm_house`, and the object name, which is `pig1`:

```
farm_house pig1,pig2,pig3;
```

On line 28, you set the value of `pig1` to `12`. This is done using the dot operator (`.`). The object `pig1` has access to the method `set()`. The `set()` method is a method of the class `farm_house`, so it has access to its private data. The implementation of the `set()` method is shown on line 13. For the program to know that the `set()` method is within the scope of the class `farm_house`, you use the scope (`::`) operator. On line 15, the variable `input` is set to the variable `pig_values`.

The class `farm_house` declared two public methods. The other method is the `get()` method. The `get()` method is implemented on line 18. The `get()` method takes no parameters but only returns the `pig_values` because it also is within the scope of the class `farm_house`.

On line 32, the `get()` method is again called by the objects `pig1`, `pig2`, and `pig3` to return the `pig_values` to the screen.

If you compare the two programs `struct.cpp` and `clasfarm.cpp`, you notice that one is about 23 lines, whereas the other is 38 lines. The code just got longer by implementing classes! This is true. The big benefits of using classes are really seen in more complex and larger programs. Also, because you hide critical data from the user, using classes is safer and less error prone. It enables the compiler to find mistakes before they become bugs.

Constructors and Destructors

Earlier, I defined the syntax of a class. In the syntax, I mentioned constructors and destructors. However, in the example `clasfarm.cpp`, you did not define any constructors or destructors. If a constructor or a destructor is not defined, the compiler creates one for you.

The Constructor Function

A *constructor* is a class initialization function that is executed automatically when a class instance is created. A constructor must abide by the following rules:

- The constructor must have the same name as its class name:

```
class farm_house
{
public:
farm_house();    //constructor
.....
.....
}
```

- The constructor cannot be defined with a return value.
- A constructor without any arguments is a default constructor.
- The constructor must be declared with the `public` keyword.

The Destructor Function

A destructor function is the opposite of a constructor function, which is executed automatically when the block in which the object is initialized is exited. A destructor releases the object and hence frees up the memory that was allocated. A destructor must abide by the following rules:

- The destructor must have the same name as the class.
- The destructor function must be preceded by ~.
- The destructor has neither arguments nor a return value.
- The destructor function must be declared with the keyword `public`.

```
class farm_house
{
```

```
public:
farm_house ();          // Constructor function
~farm_house();          // Destructor function
.....
}
```

Friend Functions and Friend Classes

Methods and members that are declared private are accessible only to that part of the program that is part of the class. However, a function outside the class or another class may be defined as a friend class or function. You can declare an entire class or individual functions as friends. You must follow some critical rules when declaring friend functions:

- The use of friends should be kept to a minimum because it overrides the benefit of hiding the data.
- Declaring `x` as a friend of `y` does not necessarily mean that `y` has access to the methods and members of `x`.

Class Declarations and Definitions

Whenever you use classes, they have their own private and public member variables and methods. As you saw in the previous `Clasfarm.cpp` example, the program is getting lengthy. There are no hard rules, but there are some standard practices followed by almost all programmers. The procedure is to put all class declarations in the header files. A header file is a file with an `.h` or `.hpp` extension. All the class definitions are placed in the `.cpp` file. The beginning of the `.cpp` file has an include directive for the header file. For example, the `clasfarm` program would be separated into `clasfarm.h` and `Clasfarm.cpp`. The `Clasfarm.h` file would look like Listing A.10.

LISTING A.10. `Clasfarm.h`.

```
 1: // Workspace:  Class2
 2: // Program Name:  Clasfarm.hpp
 3: #include <iostream.h>
 4:
 5: class farm_house
 6: {
 7:  int pig_values;
 8: public:
 9:  void set(int input);
10:  int get(void);
11: };
```

The `Clasfarm.cpp` file is in Listing A.11.

LISTING A.11. `Clasfarm.cpp`.

```
 1: #include <clasfarm.h>
 2: void farm_house::set(int input)
 3: {
 4:  pig_values = input;
 5: }
 6:
 7: int farm_house::get(void)
 8: {
 9:  return pig_values;
10: }
11:
12: int main()
13: {
14: farm_house pig1, pig2, pig3;
15:
16:
17:  pig1.set(12);
18:  pig2.set(13);
19:  pig3.set(14);
20:
21:  cout << "The value of pig1 is " << pig1.get() << "\n";
22:  cout << "The value of pig2 is " << pig2.get() << "\n";
23:  cout << "The value of pig3 is " << pig3.get() << "\n";
24:
25:  return 0;
26:
27: };
```

Classes Within a Class

It is perfectly legal to have another class declaration within a given class. This is often referred to as nesting classes. The following example declares two classes, `Lot_size` and `Tax_assessment`. The `Tax_assessment` class object `taxes` is defined within the `Lot_size` class. The `main()` method has no objects of the `Tax_assessment` class, so the methods or members of the `Tax_assessment` class cannot be directly accessed from the `main()` function. Let's review the program in Listing A.12.

LISTING A.12. Class3.cpp.

```
 1: // Workspace Name: Class3
 2: // Program Name:   Class3.cpp
 3: #include <iostream.h>
```

```
 4:
 5: class Tax_assessment
 6: {
 7:  int city_tax;
 8:  int prop_tax;
 9: public:
10:  void set(int in_city, int in_prop)
11:      {city_tax = in_city; prop_tax = in_prop; }
12:  int get_prop_tax(void) {return prop_tax;}
13:  int get_city_tax(void) {return city_tax;}
14: };
15:
16:
17: class Lot_size {
18:  int length;
19:  int width;
20:  Tax_assessment taxes;
21: public:
22:  void set(int l, int w, int s, int p) {
23:     length = l;
24:     width = w;
25:     taxes.set(s, p); }
26:  int get_area(void) {return length * width;}
27:  int get_data(void) {return taxes.get_prop_tax() ;}
28:  int get_data2(void) {return taxes.get_city_tax() ;}
29: };
30:
31:
32: int main()
33: {
34: Lot_size small, medium, large;
35:
36:  small.set(5, 5, 5, 25);
37:  medium.set(10, 10, 10, 50);
38:  large.set(20, 20, 15, 75);
39:
40:
41:  cout << "For a small lot of area "<< small.get_area ()<< "\n";
42:  cout << "the city taxes are $ "<< small.get_data2 () << "\n";
43:  cout << "and property taxes are $ " << small.get_data ()<< "\n";
44:
45:  cout << "For a medium lot of area "<< medium.get_area ()<< "\n";
46:  cout << "the city taxes are $ "<< medium.get_data2 () << "\n";
47:  cout << "and property taxes are $ " << medium.get_data ()<< "\n";
48:
49:  cout << "For a Large lot of area "<< large.get_area ()<< "\n";
50:  cout << "the city taxes are $ "<< large.get_data2 () << "\n";
51:  cout << "and property taxes are $ " << large.get_data ()<< "\n";
52:  return 0;
53: }
```

When you execute this program, it outputs the area of an rectangle and also the hypothetical taxes on rectangular area. The output is shown in Figure A.11.

FIGURE A.11.

Output from `Class3.cpp`.

In lines 5 through 14, the class `Tax_assessment` is defined. It consists of two private data members, `int city_tax` and `int prop_tax`. The class has three public methods. It is important to note the declaration and definition of these methods. In the earlier examples, you only declared the methods in the class. The function definitions were accessed using the scope (`::`) operator. In this example, you declare the method and also write its definition. This technique is referred to as inline implementation of the function. If a function definition is small and concise, this is a good technique to employ. This technique is also used to increase program efficiency (speed of execution) because the program does not have to jump in and out of a function definition.

The data members `city_tax` and `prop_tax` are private so they can only be accessed via their member methods—namely, `set()`, `get_prop_tax()`, and `get_city_tax()`.

Lines 17 through 29 declare the class `Lot_size` with its data members and methods. On line 20, the class `Tax_assessment` is embedded in this class. The object `taxes` is also declared on this line, and it is under the privacy of the class `Lot_size`. The only methods that would be able to access this object are the ones belonging to the `Lot_size` class. The `Lot_size` class has four public methods declared and defined on line 22 and lines 26 through 28. Line 25 of the `set()` method has another `set()` method defined. This is not a recursive method but rather another example of function overloading. The `set()` method on line 10 and line 22 differ in the number of parameters. The `set()` method on line 25 can access the object taxes because it is defined under the class `Tax_assessment` on line 20.

The main() function begins on line 32 and has a return type int. On line 34, the objects of class Lot_size are declared. On lines 36 through line 38, the values of the objects are set using the set() method. An important point to note is that the class Tax_assessment has no objects in the main() method, so you cannot access any data member or method of this class from main().

On line 41, the area of the Lot_size is output by operating the get_area() method on an object of class Lot_size. On line 42, the city taxes are output by operating the method get_data2 on an object of Lot_size. This approach is required because the city_tax is a member data of class Tax_assessment, which cannot be operated on directly in the main() method. You use the method get_data2, which is a method of Lot_size and has access to the object taxes, which in turn can be accessed via get_city_tax.

Inheritance

One of the advantages of programming in C++ or any other object-oriented language is taking a global to local approach. Suppose you need to develop a program that comprehends all metals and their characteristics. If you take the class approach of the previous section, you would probably have one class named metals. The data members of metals would probably be density and volume. You could have another class named gold and one for aluminum. The data members describing gold and aluminum would need all the properties of metals in addition to their own data members such as color and shine. If you could devise a hierarchy of classes such that the classes for gold and aluminum would have only their individual data members but inherit the generic properties from the parent metals class—then you would be using inheritance.

Inheritance is also called derivation. The new class inherits the functionality of an existing class. The existing class is called the base class, and the new class is called the derived class. A similar inheritance can be derived for animals, mammals, and dogs.

To derive a new class from a base class, you use the colon (:) operator:

```
class human : public mammal
```

In this example, the new class human is derived from the base class mammal. The derived class human would have all the functionality of the base class mammal. In addition, the human class can have other functionality, such as the ability to drive a car and work for food. The example in Listing A.13 shows how to create the objects of type human and access its data and functions.

LISTING A.13. Inherit1.cpp.

```
 1: // Workspace Name:  Inherit
 2: // Program Name :  Inherit1.cpp
 3:
 4: #include <iostream.h>
 5: enum GENDER { MALE, FEMALE };
 6:
 7: class Mammal
 8: {
 9: public:
10:   // constructors
11:   Mammal():itsAge(35), itsWeight(180){}
12:   ~Mammal(){}
13:
14:   int GetAge()const { return itsAge; }
15:   void SetAge(int age) { itsAge = age; }
16:   int GetWeight() const { return itsWeight; }
17:   void SetWeight(int weight) { itsWeight = weight; }
18:
19:
20: protected:
21:   int itsAge;
22:   int itsWeight;
23:  };
24:
25: class Human : public Mammal
26: {
27: public:
28:
29:   // Constructors
30:   Human():itsGender(MALE){}
31:   ~Human(){}
32:
33:   GENDER GetGender() const { return itsGender; }
34:   void SetGender(GENDER gender) { itsGender = gender; }
35:
36:   void Drive() { cout << "Driving to work...\n"; }
37:   void Work() { cout << "working...\n"; }
38:
39: private:
40:   GENDER itsGender;
41: };
42:
43: void main()
44: {
45:   Human John_doe;
46:   John_doe.Drive();
47:   John_doe.Work();
48:   cout << "John_doe is " << John_doe.GetAge() << " years old\n";
```

A

```
49: cout << "And weighs " <<John_doe.GetWeight() << " lbs \n";
50: }
```

The output from Listing A.13 is shown in Figure A.12.

FIGURE A.12.
Inheritance output.

On line 5, a new keyword `enum` is defined. Enumerate defines a new data type with a list of identifiers. The identifiers are fixed values that increment automatically. In this example, the variable `MALE` has a value `0` and the variable `FEMALE` has a value `1` by default. You could also specify a value:

```
enum Alphabets ( A, B, C=5, D=1)
```

In this case, `A` has a value `0`, `B` is `1`, `C` is `5`, and `D` is `1`. If you did not specify any values, then

`A` is `0`

`B` is `1`

`C` is `2`

`D` is `3`

On line 20, another new keyword `protected` is defined in the class `Mammal`. You covered public and private in the section on classes where all the data members were defined under the `private` keyword. When the data member is defined as private, the derived class cannot access them. For the derived classes to be able to access the data members and methods of a class, they must be defined as protected. The `protected` keyword restricts the access only to the derived classes. Another alternative is to define these methods and members as public, in which case all classes have free access. Although this is a solution, it is not a desired solution because it moves you away from encapsulation.

Line 7 declares the base class Mammal. The constructor is on line 11, and the destructor is on line 12. In classes, whenever an object of the class is created, the class constructor is called. The constructor class performs an additional function of initializing its member data, itsAge(35) and itsWeight(180). This could have been accomplished by initializing the member data in the body of the constructor, as shown in the following:

```
Mammal()
{
itsAge = 35;
itsWeight = 180;
};
```

The technique of initializing the data members in the constructor declaration (as shown on line 11 of Listing A.14) is far more efficient due to the internal initialization of classes in C++. Use this technique whenever possible because it increases code efficiency.

With derived classes, when an object is created in the derived class, the constructor of the base class is called first and then the constructor of the derived class is called. In this example, when the object John_doe is created for the first time, the constructor of the base class Mammal is called. The object John_doe is not created until both the base constructor and derived class constructor are called. With destructors, the reverse order is followed; when the object John_doe ceases to exist, the derived class destructor is called before the base class destructor. On line 25, you define the name of the derived class and its relevant base class.

Line 48 and line 49 are critical in terms of how the data is accessed and output. On lines 48 and 49, the Human object John_doe accesses information directly from the base class of Mammal. Remember from the example class3.cpp, to output data from a nested class, you had to use indirect access to the class Tax_assessment.

Inheritance is a significant tool in object-oriented programming, and if it's used effectively, it provides code reusability. The inherit1.cpp program gave you an overall flavor of inheritance and its properties. However, when programs are written in real life, they are structured more efficiently. The next program involves a more logical and formal process of writing a program.

Assume you are writing a program for the automobile market. The automobile market consists of cars, trucks, minivans, and SUVs (sport utility vehicles). automobile is the parent or base class, and the others are the derived classes. Let's start by defining the automobile class. Listing A.14 shows the code.

A

LISTING A.14. `Auto.h`.

```
 1: // Workspace name: Inherit2
 2: // Program name:  Auto.h
 3:
 4: #ifndef AUTO_H
 5: #define AUTO_H
 6:
 7: class automobile
 8: {
 9: protected:
10:  int miles_per_gallon;
11:  float fuel_capacity;
12: public:
13:  void initialize(int in_mpg, float in_fuel);
14:  int get_mpg(void);
15:  float get_fuel(void);
16:  float travel_distance(void);
17: }
18:
19: #endif
```

Lines 4 and 5 include directives to the preprocessor. The directive on line 4 is covered in detail toward the end of this section. On line 7, the class `automobile` is defined. This class has two data members and four methods. The class is included in the header file only. The definition of the methods of this class are contained in the `Auto.cpp` file in Listing A.15.

LISTING A.15. `Auto.cpp`.

```
 1: // Workspace name : Inherit2
 2: // Program name :  Auto.cpp
 3: #include "auto.h"
 4:
 5:
 6: void automobile::initialize(int in_mpg, float in_fuel)
 7: {
 8:  miles_per_gallon = in_mpg;
 9:  fuel_capacity = in_fuel;
10: }
11:
12:        // Get the rated fuel economy - miles per gallon
13: int automobile::get_mpg()
14: {
15:  return miles_per_gallon;
16: }
17:
```

continues

LISTING A.15. CONTINUED

```
18:        // Get the fuel tank capacity
19: float automobile::get_fuel()
20: {
21:  return fuel_capacity;
22: }
23:
24:        // Return the travel distance possible
25: float automobile::travel_distance()
26: {
27:  return miles_per_gallon * fuel_capacity;
28: }
```

The method `get_mpg` provides the value for the miles per gallon for a particular vehicle. The `get_fuel` method provides the gas tank capacity. Next, you define the first derived class, a `car`, in Listing A.16.

LISTING A.16. `Car.h`.

```
 1: // Workspace name: Inherit2
 2: // Program name:  Car.h
 3:
 4: #ifndef CAR_H
 5: #define CAR_H
 6:
 7: #include "auto.h"
 8:
 9: class car : public automobile
10: {
11:  int Total_doors;
12: public:
13:  void initialize(int in_mpg, float in_fuel, int door = 4);
14:  int doors(void);
15: };
16:
17: #endif
```

The class `car` is a derived class from the `automobile` class. Because it is a derived class, it has access to all of the methods of the base class `automobile`. In addition, this class has a data member for the number of doors in the car. The methods of this class are defined in `Car.cpp` in Listing A.17.

LISTING A.17. `Car.cpp`.

```
 1: // Workspace name: Inherit2
 2: // Program name:  Car.cpp
 3:
 4: #include "car.h"
 5:
 6: void car::initialize(int in_mpg, float in_fuel, int door)
 7: {
 8:  Total_doors = door;
 9:  miles_per_gallon = in_mpg;
10:  fuel_capacity = in_fuel;
11: }
12:
13:
14: int car::doors(void)
15: {
16:  return Total_doors;
17: }
```

The `initialization` method is defined in lines 6 through 11. It is important to note that the base class of the `automobile` (`auto.h`) also had an `initialization` method. The `initialization` in the `car` class overrides the base class `initialization`. Last but not least is the `main()` definition. The `main()` method is defined in `Allauto.cpp` in Listing A.18.

LISTING A.18. `Allauto.cpp`.

```
 1: // Workspace name: Inherit2
 2: // Program name:  Allauto.cpp
 3:
 4: #include <iostream.h>
 5: #include "auto.h"
 6: #include "car.h"
 7:
 8: int main()
 9: {
10:
11: car sedan;
12:
13:  sedan.initialize(24, 20.0, 4);
14:  cout << "The sedan can travel " << sedan.travel_distance() <<
15:                          " miles.\n";
16:  cout << "The sedan has " << sedan.doors() << " doors.\n";
17:
18:  return 0;
19: }
```

The `main()` definition has only one object defined. On line 11, an object of class `car` is declared. The initialization is on line 13. The `initialization` passes the fuel efficiency of the car (miles per gallon) and the tank capacity. This information is used to access the method `travel_distance` in the base class define in `auto.cpp`. The derived class has access to the methods of the base class. Additionally, the derived class passes information to its own data member about the number of doors in the vehicle. The result of executing this program is shown in Figure A.13.

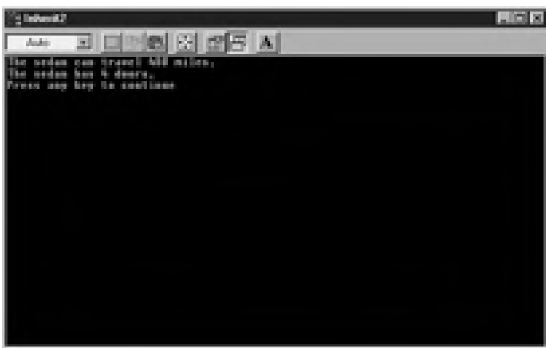


FIGURE A.13.
Vehicle class results.

You can now add more classes for other vehicle types. You can make your own classes for a truck and minivan and derive them from the base class exactly like the `car` class.

If you add another class for trucks, it is important to include the preprocessor directives from Listing A.14's lines 4 and 19. These lines are listed again in the following:

```
4    #ifndef AUTO.H
5    #define AUTO.H
.
.
.
19    #endif
```

Because the `truck` class is derived from the parent class `automobile`, it must include the file `Auto.h` in `Truck.h`. The header of the `car` class, `Car.h`, already includes `Auto.h` for the same reason. Now, if you create a method that uses both the `truck` and `car` classes, you could potentially include `Auto.h` twice, which would generate in a compiler error. To prevent this, you add lines 4 and 5 of Listing A.14. Line 4 issues a command to the compiler to verify whether the class `AUTO.H` has been defined; if it hasn't been defined, the program jumps to line 5 and defines it, and if it has been defined, the program jumps to line 19 and ends.

Summary

Congratulations! You have covered almost all of the features and properties of C++. You should now have a solid footing to take full advantage of Visual C++ and object-oriented programming.

APPENDIX B

Answers

This appendix provides the answers to the quiz questions and exercises at the end of each chapter.

Day 1

Quiz

1. How do you change the caption on a button?

   In the window layout editor, select the button to be changed. Right-click the mouse and select Properties from the pop-up menu. Change the value in the Caption field.

2. What can you do with the Visual C++ AppWizard?

   You can use it to build a shell for your application, based on the type of application and the functionality needs of the application. The shell will have support for the desired functionality already built in.

3. How do you attach functionality to the click of a button?

   By using the Class Wizard, you can create a function and attach it to an object for handling a specific Windows message. The Class Wizard creates the function and can take you right to the spot in the function's code where you need to begin adding your own code.

Exercise

Add a second button to the About window in your application. Have the button display a different message from the one on the first window.

1. In the workspace pane, select the Resource View tab.
2. Expand the dialog tree branch and double-click the `IDD_ABOUTBOX` dialog, bringing it into the Developer Studio editor.
3. Click the button control on the toolbar.
4. Click and drag the mouse on the window where you want the button to be placed.
5. Open the properties dialog for the new button, changing the ID and caption to describe the message to be displayed by the button. Close the properties dialog.
6. Open the Class Wizard and add a new function for the clicked message for your new button.
7. Click the Edit Code button in the Class Wizard to take you to the spot in your code where your new function is.
8. Add the `MessageBox` function to display a message to the user.
9. Compile and run your application to test your new button.

Day 2

Quiz

1. Why do you need to specify the tab order of the controls on your application windows?

   By specifying the tab order of the controls on your application windows, you can control the order in which the user navigates the application window. If the user is using the keyboard to navigate around the application window, then the two primary means of navigating between controls are the tab key and mnemonics that jump directly to specific controls. The tab order helps provide the user with a consistent and predictable experience when using your application.

2. How can you include a mnemonic in a static text field that will take the user to the edit box or combo box beside the text control?

   If you place a mnemonic in a static text control and then make sure that the static text control is just before the edit control associated with the static text, the user can select the mnemonic in the static text control to jump directly to the edit box control.

3. Why do you need to give unique object IDs to the static text fields in front of the edit box and combo boxes?

   The unique object IDs on the two static text controls were necessary because you need to manipulate those two controls with the check boxes that enable or disable and show or hide sets of controls.

4. Why do you need to call the `UpdateData` function before checking the value of one of the controls?

   If the user has changed the value of the control on the screen, the `UpdateData` function must be called, passing it `TRUE` as the function argument, to copy the values from the controls on the window to the variables that are associated with those controls. If `UpdateData` is not called, then the values of the variables may not correctly reflect what the user has changed on the screen.

Exercises

1. Add code to the Default Message button to reset the edit box to say `Enter a message here`.

   Using the Class Wizard, add a function to the Default Message button's clicked event. In this function, add the code in Listing B.1.

LISTING B.1. DAY2DLG.CPP—THE CODE TO PLACE A DEFAULT MESSAGE IN THE EDIT BOX.

```
 1: void CDay2Dlg::OnDfltmsg()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Set the message to a default message
10:     m_strMessage = "Enter a message here";
11:
12:     // Update the screen
13:     UpdateData(FALSE);
```

continues

LISTING B.1. CONTINUED

```
14:
15:     ///////////////////////
16:     // MY CODE ENDS HERE
17:     ///////////////////////
18: }
```

2. Add code to enable or disable and show or hide the controls used to select and run another application.

   Add functions to the Enable and Show Program Action check boxes. In these functions, add the code in Listing B.2.

LISTING B.2. DAY2DLG.CPP—THE CODE TO ENABLE OR DISABLE AND SHOW OR HIDE THE RUN PROGRAM CONTROLS.

```
 1: void CDay2Dlg::OnCkenblpgm()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the current values from the screen
10:     UpdateData(TRUE);
11:
12:     // Is the Enable Program Action check box checked?
13:     if (m_bEnablePgm == TRUE)
14:     {
15:         // Yes, so enable all controls that have anything
16:         // to do with running a program
17:         GetDlgItem(IDC_PROGTORUN)->EnableWindow(TRUE);
18:         GetDlgItem(IDC_RUNPGM)->EnableWindow(TRUE);
19:         GetDlgItem(IDC_STATICPGM)->EnableWindow(TRUE);
20:     }
21:     else
22:     {
23:         // No, so disable all controls that have anything
24:         // to do with running a program
25:         GetDlgItem(IDC_PROGTORUN)->EnableWindow(FALSE);
26:         GetDlgItem(IDC_RUNPGM)->EnableWindow(FALSE);
27:         GetDlgItem(IDC_STATICPGM)->EnableWindow(FALSE);
28:     }
29:
30:     ///////////////////////
31:     // MY CODE ENDS HERE
32:     ///////////////////////
```

```
33: }
34:
35: void CDay2Dlg::OnCkshwpgm()
36: {
37:     // TODO: Add your control notification handler code here
38:
39:     ///////////////////////
40:     // MY CODE STARTS HERE
41:     ///////////////////////
42:
43:     // Get the current values from the screen
44:     UpdateData(TRUE);
45:
46:     // Is the Show Program Action check box checked?
47:     if (m_bShowPgm == TRUE)
48:     {
49:         // Yes, so show all controls that have anything
50:         // to do with running a program
51:         GetDlgItem(IDC_PROGTORUN)->ShowWindow(TRUE);
52:         GetDlgItem(IDC_RUNPGM)->ShowWindow(TRUE);
53:         GetDlgItem(IDC_STATICPGM)->ShowWindow(TRUE);
54:     }
55:     else
56:     {
57:         // No, so hide all controls that have anything
58:         // to do with running a program
59:         GetDlgItem(IDC_PROGTORUN)->ShowWindow(FALSE);
60:         GetDlgItem(IDC_RUNPGM)->ShowWindow(FALSE);
61:         GetDlgItem(IDC_STATICPGM)->ShowWindow(FALSE);
62:     }
63:
64:     ///////////////////////
65:     // MY CODE ENDS HERE
66:     ///////////////////////
67: }
```

3. Extend the code in the `OnRunpgm` function to allow the user to enter his own program name to be run.

   Modify the `OnRunpgm` function as in Listing B.3.

LISTING B.3. DAY2DLG.CPP—THE CODE TO RUN ANY PROGRAM NAME TYPED INTO THE RUN PROGRAM COMBO BOX.

```
1: void CDay2Dlg::OnRunpgm()
2: {
3:     // TODO: Add your control notification handler code here
4:
```

continues

LISTING B.3. CONTINUED

```
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Get the current values from the screen
10:     UpdateData(TRUE);
11:
12:     // Declare a local variable for holding the program name
13:     CString strPgmName;
14:
15:     // Copy the program name to the local variable
16:     strPgmName = m_strProgToRun;
17:
18:     // Make the program name all uppercase
19:     strPgmName.MakeUpper();
20:
21:     // Did the user select to run the Paint program?
22:     if (strPgmName == "PAINT")
23:         // Yes, run the Paint program
24:         WinExec("pbrush.exe", SW_SHOW);
25:
26:     // Did the user select to run the Notepad program?
27:     if (strPgmName == "NOTEPAD")
28:         // Yes, run the Notepad program
29:         WinExec("notepad.exe", SW_SHOW);
30:
31:     // Did the user select to run the Solitaire program?
32:     if (strPgmName == "SOLITAIRE")
33:         // Yes, run the Solitaire program
34:         WinExec("sol.exe", SW_SHOW);
35:
36:         // Run any other program name typed into the combo box
37:         if ((strPgmName != "PAINT") && (strPgmName != "NOTEPAD") &&
38:             (strPgmName != "SOLITAIRE"))
39:               // Yes, run the program typed into the combo box
40:               WinExec(strPgmName, SW_SHOW);
41:
42:     ///////////////////////
43:     // MY CODE ENDS HERE
44:     ///////////////////////
45: }
```

Day 3

Quiz

1. What are the possible mouse messages that you can add functions for?

   WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_RBUTTONDOWN, WM
   _RBUTTONUP, WM_RBUTTONDBLCLK, WM_MOUSEMOVE, and WM_MOUSEWHEEL.

2. How can you tell if the left mouse button is down on the WM_MOUSEMOVE event message?

   You can mask the flags passed to the OnMouseMove function with the MK_LBUTTON flag, as follows:

   ((nFlags & MK_LBUTTON) == MK_LBUTTON)

3. How can you prevent the cursor from changing back to the default cursor after you set it to a different one?

   Return TRUE in the OnSetCursor event function, preventing the ancestor OnSetCursor function from being called.

Exercises

1. Modify your drawing program so that the left mouse button can draw in red and the right mouse button can draw in blue.

   Add a function for the WM_RBUTTONDOWN event message and write the code for it as in Listing B.4.

LISTING B.4. MOUSEDLG.CPP—THE OnRButtonDown FUNCTION.

```
 1: void CMouseDlg::OnRButtonDown(UINT nFlags, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:         // Set the current point as the starting point
10:     m_iPrevX = point.x;
11:     m_iPrevY = point.y;
12:
13:     ///////////////////////
14:     // MY CODE ENDS HERE
15:     ///////////////////////
16:
17:     CDialog::OnRButtonDown(nFlags, point);
18: }
```

Extend the `OnMouseMove` function as in Listing B.5.

LISTING B.5. MOUSEDLG.CPP—THE MODIFIED `OnMouseMove` FUNCTION.

```
 1: void CMouseDlg::OnMouseMove(UINT nFlags, CPoint point)
 2: {
 3:     // TODO: Add your message handler code here and/or call default
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:         // Check to see if the left mouse button is down
10:     if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
11:     {
12:         // Get the Device Context
13:         CClientDC dc(this);
14:
15:         // Create a new pen
16:         CPen lpen(PS_SOLID, 16, RGB(255, 0, 0));
17:
18:         // Use the new pen
19:         dc.SelectObject(&lpen);
20:
21:         // Draw a line from the previous point to the current point
22:         dc.MoveTo(m_iPrevX, m_iPrevY);
23:         dc.LineTo(point.x, point.y);
24:
25:         // Save the current point as the previous point
26:         m_iPrevX = point.x;
27:         m_iPrevY = point.y;
28:     }
29:
30:     // Check to see if the right mouse button is down
31:     if ((nFlags & MK_RBUTTON) == MK_RBUTTON)
32:     {
33:         // Get the Device Context
34:         CClientDC rdc(this);
35:
36:         // Create a new pen
37:         CPen rpen(PS_SOLID, 16, RGB(0, 0, 255));
38:
39:         // Use the new pen
40:         rdc.SelectObject(&rpen);
41:
42:         // Draw a line from the previous point to the current point
43:         rdc.MoveTo(m_iPrevX, m_iPrevY);
44:         rdc.LineTo(point.x, point.y);
45:
46:         // Save the current point as the previous point
```

```
47:         m_iPrevX = point.x;
48:         m_iPrevY = point.y;
49:     }
50:
51:     ///////////////////////
52:     // MY CODE ENDS HERE
53:     ///////////////////////
54:
55:     CDialog::OnMouseMove(nFlags, point);
56: }
```

B

2. Extend the `OnKeyDown` function to add some of the following standard cursors:
    - IDC_CROSS
    - IDC_UPARROW
    - IDC_SIZEALL
    - IDC_SIZENWSE
    - IDC_SIZENESW
    - IDC_SIZEWE
    - IDC_SIZENS
    - IDC_NO
    - IDC_APPSTARTING
    - IDC_HELP

   Your modified `OnKeyDown` function can look something like the following:

```
void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    char lsChar;         // The current character being pressed
    HCURSOR lhCursor;    // The handle to the cursor to be displayed

    // Convert the key pressed to a character
    lsChar = char(nChar);

    // Is the character "A"
    if (lsChar == 'A')
    {
        // Load the arrow cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
        // Set the cursor flag
```

```
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    // Is the character "B"
    if (lsChar == 'B')
    {
        // Load the I beam cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_IBEAM);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    // Is the character "C"
    if (lsChar == 'C')
    {
        // Load the hourglass cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_WAIT);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    // Is the character "D"
    if (lsChar == 'D')
    {
        // Load the cross hair cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    // Is the character "E"
    if (lsChar == 'E')
    {
        // Load the up arrow cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_UPARROW);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    // Is the character "F"
    if (lsChar == 'F')
```

```
{
    // Load the size cursor
    lhCursor = AfxGetApp()->LoadStandardCursor(IDC_SIZEALL);
    // Set the cursor flag
    m_bCursor = TRUE;
    // Set the screen cursor
    SetCursor(lhCursor);
}

// Is the character "G"
if (lsChar == 'G')
{
    // Load the up/right-down/left size cursor
    lhCursor = AfxGetApp()->LoadStandardCursor(IDC_SIZENWSE);
    // Set the cursor flag
    m_bCursor = TRUE;
    // Set the screen cursor
    SetCursor(lhCursor);
}

// Is the character "H"
if (lsChar == 'H')
{
    // Load the up/left-down/right size cursor
    lhCursor = AfxGetApp()->LoadStandardCursor(IDC_SIZENESW);
    // Set the cursor flag
    m_bCursor = TRUE;
    // Set the screen cursor
    SetCursor(lhCursor);
}

// Is the character "I"
if (lsChar == 'I')
{
    // Load the left-right size cursor
    lhCursor = AfxGetApp()->LoadStandardCursor(IDC_SIZEWE);
    // Set the cursor flag
    m_bCursor = TRUE;
    // Set the screen cursor
    SetCursor(lhCursor);
}

// Is the character "J"
if (lsChar == 'J')
{
    // Load the up-down size cursor
    lhCursor = AfxGetApp()->LoadStandardCursor(IDC_SIZENS);
    // Set the cursor flag
    m_bCursor = TRUE;
    // Set the screen cursor
    SetCursor(lhCursor);
}
```

```
    if (lsChar == 'K')
    {
        // Load the no cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_NO);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    if (lsChar == 'L')
    {
        // Load the app starting cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_APPSTARTING);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    if (lsChar == 'M')
    {
        // Load the help cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_HELP);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
    }

    // Is the character "X"
    if (lsChar == 'X')
    {
        // Load the arrow cursor
        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
        // Set the cursor flag
        m_bCursor = TRUE;
        // Set the screen cursor
        SetCursor(lhCursor);
        // Exit the application
        OnOK();
    }

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////

    CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

B

Day 4

Quiz

1. What did you accomplish by adding the two timer IDs to the resource symbols?

   You defined the two IDs so that they were available as constants throughout the application.

2. What is another way to add these two IDs to the application?

   Add them as `#define` constants in the class header file (`Day2Dlg.h`), as follows:

```
///////////////////////////////////////////////////////////////////////
// CTimersDlg dialog

#define ID_CLOCK_TIMER 1
#define ID_COUNT_TIMER 2

class CTimersDlg : public CDialog
{
.
.
.
```

3. How can you tell two timers apart in the `OnTimer` function?

   You use the timer ID to determine which timer triggered the event.

4. How many timer events does your application receive if the timer is set for one second and your application has been busy for one minute, preventing it from receiving any timer event messages?

   One.

Exercise

Update your application so that when the counter timer is started, the clock timer is reset to run at the same interval as the counter timer. When the counter timer is stopped, return the clock timer to a one-second interval.

To change the interval at which a timer is running, you need to first stop the timer and then restart it, as in Listing B.6.

LISTING B.6. THE REVISED `OnStarttime` AND `OnStoptimer` FUNCTIONS.

```
 1: void CTimersDlg::OnStarttime()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Update the variables
10:     UpdateData(TRUE);
11:
12:     // Initialize the count
13:     m_iCount = 0;
14:     // Format the count for displaying
15:     m_sCount.Format("%d", m_iCount);
16:
17:     // Update the dialog
18:     UpdateData(FALSE);
19:     // Start the timer
20:     SetTimer(ID_COUNT_TIMER, m_iInterval, NULL);
21:
22:     // Stop the clock timer
23:     KillTimer(ID_CLOCK_TIMER);
24:     // Restart the clock timer with the counter interval
25:     SetTimer(ID_CLOCK_TIMER, m_iInterval, NULL);
26:
27:     // Enable the Stop Timer button
28:     m_cStopTime.EnableWindow(TRUE);
29:     // Disable the Start Timer button
30:     m_cStartTime.EnableWindow(FALSE);
31:
32:     ///////////////////////
33:     // MY CODE ENDS HERE
34:     ///////////////////////
35: }
36:
37: void CTimersDlg::OnStoptimer()
38: {
39:     // TODO: Add your control notification handler code here
40:
41:     ///////////////////////
42:     // MY CODE STARTS HERE
43:     ///////////////////////
44:
45:     // Stop the timer
46:     KillTimer(ID_COUNT_TIMER);
47:
48:     // Stop the clock timer
```

```
49:     KillTimer(ID_CLOCK_TIMER);
50:     // Restart the clock timer with 1 second interval
51:     SetTimer(ID_CLOCK_TIMER, 1000, NULL);
52:
53:     // Disable the Stop Timer button
54:     m_cStopTime.EnableWindow(FALSE);
55:     // Enable the Start Timer button
56:     m_cStartTime.EnableWindow(TRUE);
57:
58:     ///////////////////////
59:     // MY CODE ENDS HERE
60:     ///////////////////////
61: }
```


Day 5

Quiz

1. What are the possible return codes that your application might receive from the `MessageBox` function call when you specify the `MB_RETRYCANCEL` button combination?

   `IDRETRY` and `IDCANCEL`.

2. What are the common dialogs that are built into the Windows operating systems that are defined as MFC classes?

   The common Windows dialogs that are defined as MFC classes are

   - File selection
   - Font selection
   - Color selection
   - Page setup for printing
   - Printing
   - Find and replace

3. What is the difference between a modal dialog and a modeless dialog?

   A modal dialog stops all application processing until the user responds to the dialog. A modeless dialog allows the user to continue working with the rest of the application while the dialog is open for use.

4. How can you display a File Save dialog for the user instead of the File Open dialog that you did have in your application?

   In the class instance variable declaration, pass `FALSE` instead of `TRUE`. This makes the variable declaration look like this:

   ```
   CFileDialog m_ldFile(FALSE);
   ```

5. Why did you not need to create any functions and add any code to your custom dialog?

   The only functionality that was needed on the custom dialog was calling `UpdateData` before closing the dialog. Because the OK and Cancel buttons were never deleted from the dialog, the OK button automatically performed this functionality.

Exercises

1. Modify your application so that it includes the directory with the filename in the application. (Hint: The `GetFileName` function returns the path and filename that was selected in the File Open dialog.)

   Modify the `OnFileopen` function as follows:

   ```
   void CDialogsDlg::OnFileopen()
   {
       // TODO: Add your control notification handler code here

       ///////////////////////
       // MY CODE STARTS HERE
       ///////////////////////

       CFileDialog m_ldFile(TRUE);

       // Show the File open dialog and capture the result
       if (m_ldFile.DoModal() == IDOK)
       {
           // Get the filename selected
           m_sResults = m_ldFile.GetPathName();
           // Update the dialog
           UpdateData(FALSE);
       }

       ///////////////////////
       // MY CODE ENDS HERE
       ///////////////////////
   }
   ```

   The `GetPathName` function returns the path and filename, so changing the function call from `GetFileName` to `GetPathName` alters the display to include the path with the filename.

2. Add a button on the custom dialog that calls the `MessageBox` function with a Yes or No selection. Pass the result back to the main application dialog.

   Follow these steps:

   1. Using the Class View, add a member variable to the `CMsgDlg` class. Specify the variable type as `int`, the name as `m_iYesNo`, and the access as Public.
   2. Using the Resource View, bring the custom dialog into the editor area. Add a command button to the window, named `IDC_YESNO` with a caption `&Yes or No`.
   3. Using the Class Wizard, add a function to the new button you just added and edit the function. Include the following code:

```
void CMsgDlg::OnYesno()
{
    // TODO: Add your control notification handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Ask the user
    m_iYesNo = MessageBox("Choose Yes or No", "Yes or No",
➥MB_YESNO);

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

   4. Add a button to the main dialog window named `IDC_YESNO` with the caption `Y&es or No`.
   5. Using the Class Wizard, add a function to the new button, including the following code:

```
void CDialogsDlg::OnYesno()
{
    // TODO: Add your control notification handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // What did the user answer
    switch (m_dMsgDlg.m_iYesNo)
    {
    case IDYES:    // Did the user answer YES?
        m_sResults = "Yes!";
        break;
```

```
        case IDNO:     // Did the user answer NO?
            m_sResults = "No!";
            break;
        }

        // Update the dialog
        UpdateData(FALSE);

        ///////////////////////
        // MY CODE ENDS HERE
        ///////////////////////
    }
```

Day 6

Quiz

1. What event message does a menu selection send to the window message queue?

   COMMAND.

2. How do you attach a menu to a dialog window?

   In the dialog designer, open the properties dialog for the window, and choose the menu from the drop-down list of menus.

3. Which existing class do you specify for handling event messages for the menu?

   The dialog class for the window on which the menu appears.

4. What event message should a pop-up menu be triggered by?

   The `WM_CONTEXTMENU` event.

Exercises

1. Add a button to the main window and have it call the same function as the Hello menu entry.

   Follow these steps:

   1. Add a button to the dialog screen. Supply a button ID of `IDC_HELLO` and a caption of `&Hello`.
   2. Using the Class Wizard, add a function to the button. Name the function `OnHello`.

2. Add a pop-up menu to your application that uses the Help drop-down menu as the pop-up menu.

Follow these steps:

1. Using the Class Wizard, add a function for the `WM_CONTEXTMENU` event message in your dialog window.
2. Edit the function, adding the following code:

```
void CMenusDlg::OnContextMenu(CWnd* pWnd, CPoint point)
{
    // TODO: Add your message handler code here and/or call
➥default

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Declare local variables
    CMenu *m_lMenu;    // A pointer to the menu
    CPoint m_pPoint;    // A copy of the mouse position

    // Copy the mouse position to a local variable
    m_pPoint = point;
    // Convert the position to a screen position
    ClientToScreen(&m_pPoint);
    // Get a pointer to the window menu
    m_lMenu - GetMenu();
    // Get a pointer to the first submenu
    m_lMenu = m_lMenu->GetSubMenu(1);
    // Show the Pop-up Menu
    m_lMenu->TrackPopupMenu(TPM_CENTERALIGN + TPM_LEFTBUTTON,
        m_pPoint.x, m_pPoint.y, this, NULL);

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////

    CDialog::OnRButtonDown(nFlags, point);
}
```

Day 7

Quiz

1. How can you specify that the text is to be underlined?

   Pass `1` as the value for the `bUnderline` argument to the `CreateFont` function.

2. How can you print your text upside down?

   Pass `1800` as the `nEscapement` argument to the `CreateFont` function.

3. How many times is the EnumFontFamProc callback function called by the operating system?

   The function is called once for each font that is available in the system, unless the callback function returns 0 and stops the listing of fonts.

Exercises

1. Add a check box to switch between using the entered text to display the font and using the font name to display the font, as in Figure 7.4.

   Add the check box to the dialog. Set its properties as follows:

   ID: IDC_CBUSETEXT

   Caption: &Use Entered Text

   Using the Class Wizard, attach a variable to this control. Specify the variable type as a boolean with the name m_bUseText.

   Using the Class Wizard, add a function for the BN_CLICKED event message for the check box. Edit the function, adding the following code:

```
void CDay7Dlg::OnCbusetext()
{
    // TODO: Add your control notification handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Update the variables with the dialog controls
    UpdateData(TRUE);
    // Using the font name for the font sample?
    if (!m_bUseText)
        // Using the font name
        m_strDisplayText = m_strFontName;
    else
        // Using the entered text
        m_strDisplayText = m_strSampText;

    // Update the dialog
    UpdateData(FALSE);

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

   Modify the OnInitDialog function to initialize the check box as follows:

```
BOOL CDay7Dlg::OnInitDialog()
{
```

```
    CDialog::OnInitDialog();
.
.
.
    // TODO: Add extra initialization here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Fill the font list box
    FillFontList();

    // Initialize the text to be entered
    m_strSampText = "Testing";
    // Copy the text to the font sample area
    m_strDisplayText = m_strSampText;
    // Initialize the check box
    m_bUseText = TRUE;
    // Update the dialog
    UpdateData(FALSE);

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////

    return TRUE;  // return TRUE  unless you set the focus
                  // to a control
}
```

Modify the `OnSelchangeLfonts` function as follows:

```
void CDay7Dlg::OnSelchangeLfonts()
{
    // TODO: Add your control notification handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Update the variables with the dialog controls
    UpdateData(TRUE);
    // Using the font name for the font sample?
    if (!m_bUseText)
    {
        // Copy the font name to the font sample
        m_strDisplayText = m_strFontName;
        // Update the dialog with the variables
        UpdateData(FALSE);
    }
    // Set the font for the sample
    SetMyFont();
```

```
    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

Finally, modify the `OnChangeEsamptext` function as follows:

```
void CDay7Dlg::OnChangeEsamptext()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    // CDialog::OnInitialUpdate()
    // function and call CRichEditCrtl().SetEventMask()
    // with the EN_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Update the variables with the dialog controls
    UpdateData(TRUE);
    // Using the text for the font sample?
    if (m_bUseText)
    {
        // Copy the current text to the font sample
        m_strDisplayText = m_strSampText;
        // Update the dialog with the variables
        UpdateData(FALSE);
    }

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

2. Add a check box to display the font sample in italics, as in Figure 7.5.

   Add the check box to the dialog. Set its properties as follows:

   `ID`: `IDC_CBITALIC`

   `Caption`: `&Italic`

   Using the Class Wizard, attach a variable to this control. Specify the variable type as a boolean with the name `m_bItalic`.

   Using the Class Wizard, add a function for the `BN_CLICKED` event message for the check box. Edit the function, adding the following code:

```
void CDay7Dlg::OnCbitalic()
{
    // TODO: Add your control notification handler code here

    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Update the variables with the dialog controls
    UpdateData(TRUE);
    // Set the font for the sample
    SetMyFont();

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```


Modify the `SetMyFont` function as in the following code:

```
void CDay7Dlg::SetMyFont()
{
    CRect rRect;        // The rectangle of the display area
    int iHeight;        // The height of the display area
    int iItalic = 0;    // Italicize the font?

    // Has a font been selected?
    if (m_strFontName != "")
    {
        // Get the dimensions of the font sample display area
        m_ctlDisplayText.GetWindowRect(&rRect);
        // Calculate the area height
        iHeight = rRect.top - rRect.bottom;
        // Make sure the height is positive
        if (iHeight < 0)
            iHeight = 0 - iHeight;
        // Should the font be italicized?
        If (m_bItalic)
            iItalic = 1;
        // Create the font to be used
        m_fSampFont.CreateFont((iHeight - 5), 0, 0, 0,
                FW_NORMAL, iItalic, 0, 0, DEFAULT_CHARSET,
                OUT_CHARACTER_PRECIS, CLIP_CHARACTER_PRECIS,
                DEFAULT_QUALITY, DEFAULT_PITCH ¦
                FF_DONTCARE, m_strFontName);

        // Set the font for the sample display area
        m_ctlDisplayText.SetFont(&m_fSampFont);
    }
}
```

Day 8

Quiz

1. What are the three values that are combined to specify a color?

   Red, green, and blue.

2. What do you use to draw on windows without needing to know what graphics card the user has?

   The device context.

3. What size bitmap can you use to make a brush from it?

   8 pixels by 8 pixels.

4. What event message is sent to a window to tell it to redraw itself?

   The `WM_PAINT` message.

5. How can you cause a window to repaint itself?

   Use the `Invalidate` function on it.

Exercises

1. Make the second dialog window resizable, and make it adjust the figures drawn on it whenever it's resized.

   Open the second dialog in the dialog layout designer. Open the properties for the window. Select the Style tab. Change the border to Resizing. Open the Class Wizard and add an event-handler function for the `WM_SIZE` event message. Edit the function that you just created and call the `Invalidate` function, as in Listing B.7.

LISTING B.7. THE `OnSize` FUNCTION.

```
1: void CPaintDlg::OnSize(UINT nType, int cx, int cy)
2: {
3:     CDialog::OnSize(nType, cx, cy);
4:
5:     // TODO: Add your message handler code here
6:     // Redraw the window
7:     Invalidate();
8:}
```

2. Add a bitmap brush to the set of brushes used to create the rectangles and ellipses.

   Open the Resources View tab on the workspace pane. Right-click on the top folder of the resource tree. Select Insert from the pop-up menu. Select Bitmap from the

list of available resources to add. Paint a pattern on the bitmap that you just created. Right-click on the bitmap ID in the workspace pane. Open the properties dialog and change the object ID to `IDB_BITMAPBRUSH`. Open the source code for the `DrawRegion` function. Add the code in Listing B.8, lines 22 through 24 and lines 105 through 112. Increase the number of loops in the `for` statement on line 39.

B

LISTING B.8. THE `DrawRegion` FUNCTION.

```
1:  void CPaintDlg::DrawRegion(CPaintDC *pdc, int iColor, int iTool, int
    ➥iShape)
2:  {
3:      // Declare and create the pens
.
.
.
19:     CBrush lVertBrush(HS_VERTICAL, m_crColors[iColor]);
20:     CBrush lNullBrush(RGB(192, 192, 192));
21:
22:     CBitmap lBitmap;
23:     lBitmap.LoadBitmap(IDB_BITMAPBRUSH);
24:     CBrush lBitmapBrush(&lBitmap);
25:
26:     // Calculate the size of the drawing regions
27:     CRect lRect;
28:     GetClientRect(lRect);
.
.
.
37:     int i;
38:     // Loop through all of the brushes and pens
39:     for (i = 0; i < 8; i++)
40:     {
41:         switch (i)
42:         {
.
.
.
103:                pdc->SelectObject(&lVertBrush);
104:                break;
105:            case 7:    // Null - Bitmap
106:                // Determine the location for this figure.
107:                lDrawRect.left = lDrawRect.left + liHorz;
108:                lDrawRect.right = lDrawRect.left + liWidth;
109:                // Select the appropriate pen and brush
110:                pdc->SelectObject(&lNullPen);
111:                pdc->SelectObject(&lBitmapBrush);
112:                break;
113:        }
```

continues

LISTING B.8. CONTINUED

```
114:        // Which tool are we using?
.
.
.
126:    pdc->SelectObject(lOldBrush);
127:    pdc->SelectObject(lOldPen);
128:}
```

Day 9

Quiz

1. How does an ActiveX container call methods in an ActiveX control?

   By using the `IDispatch` interface, the container can call the `Invoke` method, passing the DISPID of the control's method that the container wants to run.

2. How does an ActiveX control trigger events in the container application?

   The container application has its own `IDispatch` interface, through which the control can trigger events.

3. What AppWizard option must be selected for ActiveX controls to work properly in a Visual C++ application?

   You select the ActiveX Controls check box in the second step of the AppWizard.

4. How does Visual C++ make it easy to work with ActiveX controls?

   It generates C++ classes that encapsulate the control's functionality.

5. Why might it be difficult to work with older controls in Visual C++?

   Older controls might not contain the information necessary for Visual C++ to generate the C++ classes that are used to encapsulate the control's functionality.

Exercise

Modify the application so that the user can double-click a column header and make it the first column in the grid.

Using the Class Wizard, add a function to the `DblClick` event message for the grid control.

Edit the function in exercise 1 to add the following code:

```
void CActiveXDlg::OnDblClickMsfgrid()
{
    // TODO: Add your control notification handler code here
```

B

```
    ///////////////////////
    // MY CODE STARTS HERE
    ///////////////////////

    // Did the user click on a data row and not the
    // header row?
    if (m_ctlFGrid.GetMouseRow() != 0)
    {
        // If so, then zero out the column variable
        // and exit
        m_iMouseCol = 0;
        return;
    }
    // Save the column clicked on
    m_iMouseCol = m_ctlFGrid.GetMouseCol();
    // If the selected column was the first column,
    // there's nothing to do
    if (m_iMouseCol == 0)
        return;
    // Turn the control redraw off
    m_ctlFGrid.SetRedraw(FALSE);
    // Change the selected column position
    m_ctlFGrid.SetColPosition(m_iMouseCol, 0);
    // Resort the grid
    DoSort();
    // Turn redraw back on
    m_ctlFGrid.SetRedraw(TRUE);

    ///////////////////////
    // MY CODE ENDS HERE
    ///////////////////////
}
```

Day 10

Quiz

1. What does SDI stand for?

   Single Document Interface.

2. What functionality is in the view class?

   The view class is responsible for displaying the document for the user.

3. What function is called to redraw the document if the window has been hidden behind another window?

   The `OnDraw` function in the view class is called to redraw the document.

4. Where do you place code to clear out the current document before starting a new document?

   The `DeleteContents` function in the document class is where you place code to clear the current document.

5. What is the purpose of the document class?

   The document class is where the data is managed and manipulated. It maintains the abstract representation of the document being edited and processed.

Exercise

Add another pull-down menu to control the width of the pen used for drawing. Give it the following settings:

Menu Entry	*Width Setting*
Very Thin	1
Thin	8
Medium	16
Thick	24
Very Thick	32

Follow these steps:

1. Select the `CLine` class in the Class View tab of the workspace pane. Right-click the mouse and select Add Member Variable from the pop-up menu.
2. Specify the variable type as `UINT`, the name as `m_nWidth`, and the access as private. Click OK to add the variable.
3. Right-click the `CLine` constructor in the Class View tree. Select Go to Declaration from the pop-up menu.
4. Add `UINT nWidth` as a fourth argument to the constructor declaration.
5. Right-click the `CLine` constructor in the Class View tree. Select Go to Definition from the pop-up menu.
6. Modify the constructor to add the fourth argument and to set the `m_nWidth` member to the new argument, as in Listing B.9.

LISTING B.9. THE MODIFIED `CLine` CONSTRUCTOR.

```
1: CLine::CLine(CPoint ptFrom, CPoint ptTo, COLORREF crColor, UINT nWidth)
2: {
3:     //Initialize the from and to points
```

```
 4:     m_ptFrom = ptFrom;
 5:     m_ptTo = ptTo;
 6:     m_crColor = crColor;
 7:     m_nWidth = nWidth;
 8: }
```

7. Scroll down to the `Draw` function and modify it as in Listing B.10.

LISTING B.10. THE MODIFIED `Draw` FUNCTION.

```
 1: void CLine::Draw(CDC * pDC)
 2: {
 3:     // Create a pen
 4:     CPen lpen (PS_SOLID, m_nWidth, m_crColor);
 5:
 6:     // Set the new pen as the drawing object
 7:     CPen* pOldPen = pDC->SelectObject(&lpen);
 8:     // Draw the line
 9:     pDC->MoveTo(m_ptFrom);
10:     pDC->LineTo(m_ptTo);
11:     // Reset the previous pen
12:     pDC->SelectObject(pOldPen);
13: }
```

8. Scroll down to the `Serialize` function and modify it as in Listing B.11.

LISTING B.11. THE MODIFIED `Serialize` FUNCTION.

```
1: void CLine::Serialize(CArchive &ar)
2: {
3:     CObject::Serialize(ar);
4:
5:     if (ar.IsStoring())
6:         ar << m_ptFrom << m_ptTo << (DWORD) m_crColor << m_nWidth;
7:     else
8:         ar >> m_ptFrom >> m_ptTo >> (DWORD) m_crColor >> m_nWidth;
9: }
```

9. Select the `CDay10Doc` class in the Class View tab on the workspace pane. Right-click the mouse and choose Add Member Variable from the pop-up menu.
10. Specify the variable type as `UINT`, the name as `m_nWidth`, and the access as private. Click OK to add the variable.
11. Open the CDay10Doc source code (`Day10Doc.cpp`), scroll down to the `OnNewDocument` function, and edit it as in Listing B.12.

LISTING B.12. THE MODIFIED `OnNewDocument` FUNCTION.

```
 1: BOOL CDay10Doc::OnNewDocument()
 2: {
 3:     if (!CDocument::OnNewDocument())
 4:         return FALSE;
 5:
 6:     // TODO: add reinitialization code here
 7:     // (SDI documents will reuse this document)
 8:
 9:     ///////////////////////
10:     // MY CODE STARTS HERE
11:     ///////////////////////
12:
13:     // Initialize the color to black
14:     m_nColor = ID_COLOR_BLACK - ID_COLOR_BLACK;
15:     // Initialize the width to thin
16:     m_nWidth = ID_WIDTH_VTHIN - ID_WIDTH_VTHIN;
17:
18:     ///////////////////////
19:     // MY CODE ENDS HERE
20:     ///////////////////////
21:
22:     return TRUE;
23: }
```

12. Scroll down to the `AddLine` function, and modify it as in Listing B.13.

LISTING B.13. THE MODIFIED `AddLine` FUNCTION.

```
 1: CLine * CDay10Doc::AddLine(CPoint ptFrom, CPoint ptTo)
 2: {
 3:     static UINT nWidths[5] = { 1, 8, 16, 24, 32};
 4:
 5:     // Create a new CLine object
 6:     CLine *pLine = new CLine(ptFrom, ptTo,
        ➥m_crColors[m_nColor], nWidths[m_nWidth]);
 7:     try
 8:     {
 9:         // Add the new line to the object array
10:         m_oaLines.Add(pLine);
11:         // Mark the document as dirty
12:         SetModifiedFlag();
13:     }
14:     // Did we run into a memory exception?
15:     catch (CMemoryException* perr)
```

```
16:     {
17:         // Display a message for the user, giving him or her the
18:         // bad news
19:         AfxMessageBox("Out of memory", MB_ICONSTOP ¦ MB_OK);
20:         // Did we create a line object?
21:         if (pLine)
22:         {
23:             // Delete it
24:             delete pLine;
25:             pLine = NULL;
26:         }
27:         // Delete the exception object
28:         perr->Delete();
29:     }
30:     return pLine;
31: }
```

B

13. Add a new member function to the `CDay10Doc` class. Specify the function type as `UINT`, the declaration as `GetWidth`, and the access as public.
14. Edit the `GetWidth` function, adding the code in Listing B.14.

LISTING B.14. THE `GetWidth` FUNCTION.

```
1: UINT CDay10Doc::GetWidth()
2: {
3:     // Return the current width
4:     return ID_WIDTH_VTHIN + m_nWidth;
5: }
```

15. Select the Resource View tab in the workspace pane. Expand the tree so that you can see the contents of the Menu folder. Double-click the menu resource.
16. Grab the blank top-level menu (at the right end of the menu bar) and drag it to the left, dropping it in front of the View menu entry.
17. Open the properties for the blank menu entry. Specify the caption as `&Width`. Close the properties dialog.
18. Add submenu entries below the Width top-level menu. Specify the submenus in order, setting their properties as specified in Table B.1.

TABLE B.1. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	ID	ID_WIDTH_VTHIN
	Caption	&Very Thin
Menu Entry	ID	ID_WIDTH_THIN
	Caption	Thi&n
Menu Entry	ID	ID_WIDTH_MEDIUM
	Caption	&Medium
Menu Entry	ID	ID_WIDTH_THICK
	Caption	Thic&k
Menu Entry	ID	ID_WIDTH_VTHICK
	Caption	Very &Thick

19. Open the Class Wizard. Select the `CDay10Doc` in the Class Name combo box.
20. Add functions for both the `COMMAND` and `UPDATE_COMMAND_UI` event messages for all the width menu entries.
21. After you add the final menu entry function, click Edit Code.
22. Edit the Very Thin menu functions as in Listing B.15.

LISTING B.15. THE VERY THIN MENU FUNCTIONS.

```
 1: void CDay10Doc::OnWidthVthin()
 2: {
 3:     // TODO: Add your command handler code here
 4:
 5:     ///////////////////////
 6:     // MY CODE STARTS HERE
 7:     ///////////////////////
 8:
 9:     // Set the current width to Very Thin
10:     m_nColor = ID_WIDTH_VTHIN - ID_WIDTH_VTHIN;
11:
12:     ///////////////////////
13:     // MY CODE ENDS HERE
14:     ///////////////////////
15: }
16:
17: void CDay10Doc::OnUpdateWidthVthin(CCmdUI* pCmdUI)
18: {
19:     // TODO: Add your command update UI handler code here
20:
```

```
21:     //////////////////////
22:     // MY CODE STARTS HERE
23:     //////////////////////
24:
25:     // Determine if the Very Thin menu entry should be checked
26:     pCmdUI->SetCheck(GetWidth() == ID_WIDTH_VTHIN ? 1 : 0);
27:
28:     //////////////////////
29:     // MY CODE ENDS HERE
30:     //////////////////////
31: }
```

B

23. Edit the Thin menu functions as in Listing B.16. Edit the remaining menu functions in the same way, substituting their menu IDs for `ID_WIDTH_THIN`.

LISTING B.16. THE THIN MENU FUNCTIONS.

```
 1: void CDay10Doc::OnWidthThin()
 2: {
 3:     // TODO: Add your command handler code here
 4:
 5:     //////////////////////
 6:     // MY CODE STARTS HERE
 7:     //////////////////////
 8:
 9:     // Set the current width to Thin
10:     m_nColor = ID_WIDTH_THIN - ID_WIDTH_VTHIN;
11:
12:     //////////////////////
13:     // MY CODE ENDS HERE
14:     //////////////////////
15: }
16:
17: void CDay10Doc::OnUpdateWidthThin(CCmdUI* pCmdUI)
18: {
19:     // TODO: Add your command update UI handler code here
20:
21:     //////////////////////
22:     // MY CODE STARTS HERE
23:     //////////////////////
24:
25:     // Determine if the Thin menu entry should be checked
26:     pCmdUI->SetCheck(GetWidth() == ID_WIDTH_THIN ? 1 : 0);
27:
28:     //////////////////////
29:     // MY CODE ENDS HERE
30:     //////////////////////
31: }
```

Day 11

Quiz

1. What are the five base classes that are used in MDI applications?

   The `CWinApp`-derived class, the `CMDIFrameWnd`-derived class, the `CMDIChildWnd`-derived class, the `CDocument`-derived class, and the `CView`-derived class.

2. Why do you have to place the `ON_COMMAND_RANGE` message map entry outside the section maintained by the Class Wizard?

   The Class Wizard doesn't understand the `ON_COMMAND_RANGE` message map entry and thus would either remove or corrupt it.

3. What argument does `ON_COMMAND_RANGE` pass to the event function?

   The ID of the event message.

4. What event message should you use to display a pop-up menu?

   `WM_CONTEXTMENU`.

Exercise

Add the pull-down and context menus for the width, using the same pen widths as yesterday.

Follow these steps:

1. Add the Width handling code as in yesterday's exercise.
2. Add the Width menu entries using the same settings as yesterday.
3. Open the `Day11Doc.h` header file.
4. Scroll down toward the bottom of the header file until you find the protected section where the `AFX_MSG` message map is declared (search for `//{{AFX_MSG(CDay11Doc)`).
5. Add the function declarations in Listing B.17 before the line that you searched for. (The string that you searched for is the beginning marker for the Class Wizard maintained message map. Anything you place between it and the end marker, `//}}AFX_MSG`, is likely to be removed or corrupted by the Class Wizard.)

LISTING B.17. THE EVENT-HANDLER DECLARATIONS IN `Day11Doc.H`.

```
.
.
.
1: #ifdef _DEBUG
2:     virtual void AssertValid() const;
3:     virtual void Dump(CDumpContext& dc) const;
```

```
 4: #endif
 5:
 6: protected:
 7:
 8: // Generated message map functions
 9: protected:
10:     afx_msg void OnColorCommand(UINT nID);
11:     afx_msg void OnWidthCommand(UINT nID);
12:     afx_msg void OnUpdateColorUI(CCmdUI* pCmdUI);
13:     afx_msg void OnUpdateWidthUI(CCmdUI* pCmdUI);
14:     //{{AFX_MSG(CDay11Doc)
15:         // NOTE - the ClassWizard will add and remove member functions
            //here.
16:         //    DO NOT EDIT what you see in these blocks of generated
            //code !
17:     //}}AFX_MSG
18:     DECLARE_MESSAGE_MAP()
19: private:
20:     UINT m_nColor;
21:     CObArray m_oaLines;
22: };
```

6. Open the `Day11Doc.cpp` source-code file.
7. Search for the line `BEGIN_MESSAGE_MAP` and add the lines in Listing B.18 just after it. It's important that this code be between the `BEGIN_MESSAGE_MAP` line and the `//{{AFX_MSG_MAP` line. If these commands are between the `//{{AFX_MSG_MAP` and `//}}AFX_MSG_MAP` lines, then the Class Wizard will remove or corrupt them.

LISTING B.18. THE EVENT-HANDLER MESSAGE MAP ENTRIES IN `Day11Doc.cpp`.

```
 1:///////////////////////////////////////////////////////////////////////
 2: // CDay11Doc
 3:
 4: IMPLEMENT_DYNCREATE(CDay11Doc, CDocument)
 5:
 6: BEGIN_MESSAGE_MAP(CDay11Doc, CDocument)
 7:     ON_COMMAND_RANGE(ID_COLOR_BLACK, ID_COLOR_WHITE, OnColorCommand)
 8:     ON_COMMAND_RANGE(ID_WIDTH_VTHIN, ID_WIDTH_VTHICK, OnWidthCommand)
 9:     ON_UPDATE_COMMAND_UI_RANGE(ID_COLOR_BLACK, ID_COLOR_WHITE,
        ➥OnUpdateColorUI)
10:     ON_UPDATE_COMMAND_UI_RANGE(ID_WIDTH_VTHIN, ID_WIDTH_VTHICK,
        ➥OnUpdateWidthUI)
11:     //{{AFX_MSG_MAP(CDay11Doc)
12:         // NOTE - the ClassWizard will add and remove mapping macros
            //here.
```

continues

LISTING B.18. CONTINUED

```
13:         //    DO NOT EDIT what you see in these blocks of generated
            //code!
14:     //}}AFX_MSG_MAP
15: END_MESSAGE_MAP()
16:
17: const COLORREF CDay11Doc::m_crColors[8] = {
18:     RGB(   0,   0,   0),    // Black
19:     RGB(   0,   0, 255),    // Blue
.
.
.
```

8. Scroll to the bottom of the file and add the two event message handler functions in Listing B.19.

LISTING B.19. THE WIDTH MENU EVENT HANDLER FUNCTIONS.

```
 1: void CDay11Doc::OnWidthCommand(UINT nID)
 2: {
 3:     // Set the current width
 4:     m_nWidth = nID - ID_WIDTH_VTHIN;
 5: }
 6:
 7: void CDay11Doc::OnUpdateWidthUI(CCmdUI* pCmdUI)
 8: {
 9:     // Determine if the menu entry should be checked
10:     pCmdUI->SetCheck(GetWidth() == pCmdUI->m_nID ? 1 : 0);
11: }
```

9. Open the `IDR_CONTEXTMENU` in the Menu Designer.
10. In the Width cascading menu, add the width menu entries just like you did for the `IDR_DAY11TYPE` menu, using the same property settings. You can select the ID from the drop-down list of IDs if you would rather search for them instead of type.

Day 12

Quiz

1. How do you tie a toolbar button to a menu entry that triggers that same function?

   Give the toolbar button the same object ID as the menu entry.
2. How do you make sure that a toolbar can be docked with the frame window?

   Both must have docking enabled on the same sides (using the `EnableDocking` function) in the `OnCreate` function of the frame class.

3. How can you remove the Num Lock status indicator from the status bar?

   Remove the `ID_INDICATOR_NUM` from the indicators table near the top of the main frame source code file.

4. Why do you have to edit the resource file to add a combo box to a toolbar?

   You need to add a separator to the toolbar as a placeholder in the toolbar. The toolbar designer will do its best to prevent you from adding the separators, assuming that they are a mistake.

Exercises

1. Add another pane to the status bar to display the current width selected.

   Add an entry to the strings table with an ID of `ID_INDICATOR_WIDTH` and a caption of `VERY THICK`.

   Add another entry to the status bar indicators table at the beginning of `CMainFrame.cpp`:

```
static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_WIDTH,
    ID_INDICATOR_COLOR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

   Add a new member function to the `CToolbarDoc` class. Specify the function type as `afx_msg void`, the function definition as `OnUpdateIndicatorWidth (CCmdUI *pCmdUI)`, and the access as protected. Edit the function as follows:

```
void CToolbarDoc::OnUpdateIndicatorWidth(CCmdUI *pCmdUI)
{
    CString strWidth;

    // What is the current width?
    switch (m_nWidth)
    {
    case 0:    // Very Thin
        strWidth = "VERY THIN";
        break;
    case 1:    // Thin
        strWidth = "THIN";
        break;
    case 2:    // Medium
        strWidth = "MEDIUM";
        break;
    case 3:    // Thick
        strWidth = "THICK";
        break;
```

```
    case 4:    // Very Thick
        strWidth = "VERY THICK";
        break;
    }
    // Enable the status bar pane
    pCmdUI->Enable(TRUE);
    // Set the text of the status bar pane
    // to the current width
    pCmdUI->SetText(strWidth);
}
```

Edit the `CToolbarDoc` message map, adding the `ON_UPDATE_COMMAND_UI` message handler entry as follows:

```
/////////////////////////////////////////////////////////////////
// CToolbarDoc

IMPLEMENT_DYNCREATE(CToolbarDoc, CDocument)

BEGIN_MESSAGE_MAP(CToolbarDoc, CDocument)
    ON_UPDATE_COMMAND_UI(ID_INDICATOR_WIDTH,
➥OnUpdateIndicatorWidth)
    ON_UPDATE_COMMAND_UI(ID_INDICATOR_COLOR,
➥OnUpdateIndicatorColor)
    //{{AFX_MSG_MAP(CToolbarDoc)
    ON_UPDATE_COMMAND_UI(ID_WIDTH_VTHIN, OnUpdateWidthVthin)
.
.
    ON_COMMAND(ID_WIDTH_VTHIN, OnWidthVthin)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

2. Add a button to the main toolbar that can be used to toggle the color toolbar on and off, as in Figure 12.7.

   Open the `IDR_MAINFRAME` toolbar in the toolbar designer. Paint an icon for the blank button at the end of the toolbar. Double-click the button to open its properties dialog. Specify the button ID as `ID_VIEW_COLORBAR` and enter an appropriate prompt for the button. Recompile and run your application, and the color toolbar toggle should be working on the main toolbar.

Day 13

Quiz

1. What two macros do you have to add to a class to make it serializable?

   `DECLARE_SERIAL` and `IMPLEMENT_SERIAL`.

2. How can you determine whether the `CArchive` object is reading from or writing to the archive file?

You call the `IsStoring` or `IsLoading` functions.

3. What arguments do you need to pass to the `IMPLEMENT_SERIAL` macro?

   The class name, the base class name, and the version number.

4. What class do you need to inherit the view class from to be able to use the dialog designer to create a form for the main window in an SDI or MDI application?

   `CFormView`.

5. What type of file does the `CArchive` write to by default?

   `CFile`.

Exercise

Add a couple of radio buttons to the form to specify the person's sex, as in Figure 13.5. Incorporate this change into the `CPerson` class to make the field persistent.

In the window designer, add the two radio buttons and the static text prompt. Specify the control properties in Table B.2.

TABLE B.2. CONTROL PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`Sex:`
Radio Button	`ID`	`IDC_RMALE`
	`Caption`	`Mal&e`
	`Group`	`Checked`
Radio Button	`ID`	`IDC_RFEMALE`
	`Caption`	`&Female`

Move the mnemonic in the First button from `&First` to `Fi&rst` to prevent a conflict with the new radio buttons.

Attach a variable to the new radio buttons as in Table B.3.

TABLE B.3. CONTROL VARIABLES.

Object	*Name*	*Category*	*Type*
`IDC_RMALE`	`m_iSex`	`Value`	`int`

Increment the version number in the `IMPLEMENT_SERIAL` macro in the `CPerson` class. Add a new member variable to the `CPerson` class. Specify the type as `int`, the name as `m_iSex`, and the access as private. Update the `CPerson` constructor function, adding the `m_iSex` variable to the initializations as in line 8 of Listing B.20.

LISTING B.20. THE MODIFIED `CPerson` CONSTRUCTOR.

```
 1: CPerson::CPerson()
 2: {
 3:     // Initialize the class variables
 4:     m_iMaritalStatus = 0;
 5:     m_iAge = 0;
 6:     m_bEmployed = FALSE;
 7:     m_sName = "";
 8:     m_iSex = 0;
 9: }
```

Add the inline functions to the `CPerson` class declaration to set and get the value of this new variable, as in lines 9 and 15 of Listing B.21.

LISTING B.21. THE MODIFIED `CPerson` CLASS DECLARATION.

```
 1: class CPerson : public CObject
 2: {
 3:     DECLARE_SERIAL (CPerson)
 4: public:
 5:     // Functions for setting the variables
 6:     void SetEmployed(BOOL bEmployed) { m_bEmployed = bEmployed;}
 7:     void SetMaritalStat(int iStat) { m_iMaritalStatus = iStat;}
 8:     void SetAge(int iAge) { m_iAge = iAge;}
 9:     void SetSex(int iSex) { m_iSex = iSex;}
10:     void SetName(CString sName) { m_sName = sName;}
11:     // Functions for getting the current settings of the variables
12:     BOOL GetEmployed() { return m_bEmployed;}
13:     int GetMaritalStatus() { return m_iMaritalStatus;}
14:     int GetAge() {return m_iAge;}
15:     int GetSex() {return m_iSex;}
16:     CString GetName() {return m_sName;}
17:     CPerson();
18:     virtual ~CPerson();
19:
20: private:
21:     BOOL m_bEmployed;
22:     int m_iMaritalStatus;
23:     int m_iAge;
24:     CString m_sName;
25: };
```

Update the `Serialize` function in the `CPerson` class to include the `m_iSex` variable as in lines 9 and 12 of Listing B.22.

LISTING B.22. THE MODIFIED `CPerson.Serialize` FUNCTION.

```
 1: void CPerson::Serialize(CArchive &ar)
 2: {
 3:     // Call the ancestor function
 4:     CObject::Serialize(ar);
 5:
 6:     // Are we writing?
 7:     if (ar.IsStoring())
 8:         // Write all of the variables, in order
 9:         ar << m_sName << m_iAge << m_iMaritalStatus << m_bEmployed <<
    ➥m_iSex;
10:     else
11:         // Read all of the variables, in order
12:         ar >> m_sName >> m_iAge >> m_iMaritalStatus >> m_bEmployed >>
    ➥m_iSex;
13:
14: }
```

Modify the `PopulateView` function in the view object to include the `Sex` variable in the data exchange, as in line 19 of Listing B.23.

LISTING B.23. THE MODIFIED `CSerializeView.PopulateView` FUNCTION.

```
 1: void CSerializeView::PopulateView()
 2: {
 3:     // Get a pointer to the current document
 4:     CSerializeDoc* pDoc = GetDocument();
 5:     if (pDoc)
 6:     {
 7:         // Display the current record position in the set
 8:         m_sPosition.Format("Record %d of %d", pDoc->GetCurRecordNbr(),
 9:                 pDoc->GetTotalRecords());
10:     }
11:     // Do we have a valid record object?
12:     if (m_pCurPerson)
13:     {
14:         // Yes, get all of the record values
15:         m_bEmployed = m_pCurPerson->GetEmployed();
16:         m_iAge = m_pCurPerson->GetAge();
17:         m_sName = m_pCurPerson->GetName();
18:         m_iMaritalStatus = m_pCurPerson->GetMaritalStatus();
19:         m_iSex = m_pCurPerson->GetSex();
20:     }
21:     // Update the display
22:     UpdateData(FALSE);
23: }
```

Add an event handler for the clicked event of both new radio buttons, using the same function for both event handlers. Update the record object's field using the `Set` function, as in Listing B.24.

LISTING B.24. THE `CSerializeView.OnSex` FUNCTION.

```
 1: void CSerializeView::OnSex()
 2: {
 3:     // TODO: Add your control notification handler code here
 4:
 5:     // Sync the data in the form with the variables
 6:     UpdateData(TRUE);
 7:     // If we have a valid person object, pass the data changes to it
 8:     if (m_pCurPerson)
 9:         m_pCurPerson->SetSex(m_iSex);
10:     // Get a pointer to the document
11:     CSerializeDoc * pDoc = GetDocument();
12:     if (pDoc)
13:         // Set the modified flag in the document
14:         pDoc->SetModifiedFlag();
15: }
```

Day 14

Quiz

1. What does ODBC stand for?

   Open Database Connector.

2. What functions can you use to navigate the record set in a `CRecordset` object?

   `Move`, `MoveNext`, `MovePrev`, `MoveFirst`, `MoveLast`, and `SetAbsolutePosition`.

3. What view class should you use with an ODBC application?

   `CRecordView`.

4. What sequence of functions do you need to call to add a new record to a record set?

   `AddNew`, `Update`, and `Requery`.

5. What function do you need to call before the fields in the `CRecordset` object can be updated with any changes?

   `Edit`.

Exercise

Add a menu entry and dialog to let the user indicate the record number to move to, and then move to that record.

1. Create a new dialog, designing the dialog layout as in Figure B.1. Configure the controls as in Table B.4.

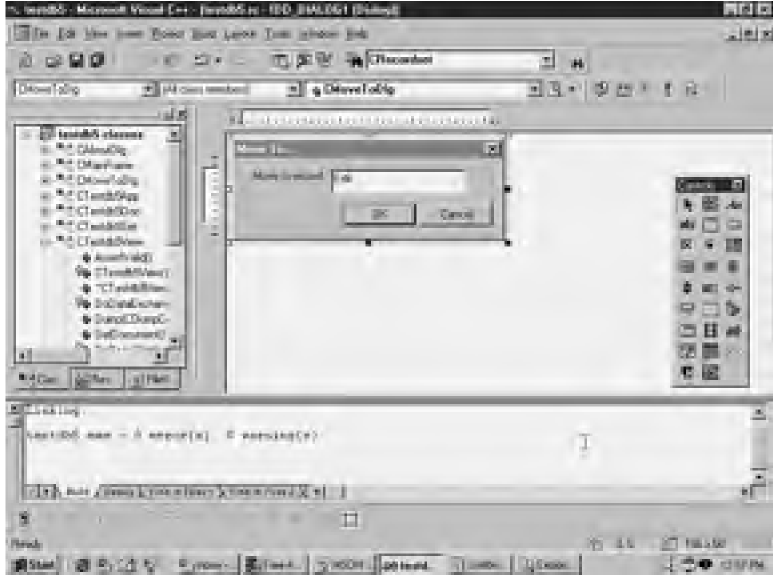


FIGURE B.1. *The Move To dialog layout.*

TABLE B.4. DIALOG PROPERTY SETTINGS.

Object	*Property*	*Setting*
Static Text	`ID`	`IDC_STATIC`
	`Caption`	`Move to record:`
Edit Box	`ID`	`IDC_ERECNBR`

2. Open the Class Wizard. Create a new class for the new dialog. Give the new class the name `CMoveToDlg`. After you create the new class, add a variable to the Edit Box control. Specify the variable type as long and the name as `m_lRowNbr`.
3. Add another menu entry to the main application menu. Specify the menu properties as in Table B.5.

TABLE B.5. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	ID	IDM_RECORD_MOVE
	Caption	&Move To...
	Prompt	Move to a specific record\nMove To

4. Open the Class Wizard and add an event-handler function for the COMMAND message for this new menu to the view class. Edit this function, adding the code in Listing B.25.

LISTING B.25. THE CDbOdbcView OnRecordMove FUNCTION.

```
 1:  void CTestdb5View::OnRecordMove()
 2:  {
 3:      // TODO: Add your command handler code here
 4:      // Create an instance of the Move To dialog
 5:      CMoveToDlg dlgMoveTo;
 6:      // Get the row number to move to
 7:      if (dlgMoveTo.DoModal() == IDOK)
 8:      {
 9:          // Get a pointer to the record set
10:         CRecordset* pSet = OnGetRecordset();
11:         // Make sure that there are no outstanding changes to be saved
12:         if (pSet->CanUpdate() && !pSet->IsDeleted())
13:         {
14:             pSet->Edit();
15:             if (!UpdateData())
16:                 return;
17:
18:             pSet->Update();
19:         }
20:         // Set the new position
21:         pSet->SetAbsolutePosition(dlgMoveTo.m_lRowNbr);
22:         // Update the form
23:         UpdateData(FALSE);
24:     }
25: }
```

5. Include the header file for the new dialog in the view class source code, as in line 10 of Listing B.26.

LISTING B.26. THE `CDbOdbcView` INCLUDES.

```
 1:  // DbOdbcView.cpp : implementation of the CDbOdbcView class
 2:  //
 3:
 4:  #include "stdafx.h"
 5:  #include "DbOdbc.h"
 6:
 7:  #include "DbOdbcSet.h"
 8:  #include "DbOdbcDoc.h"
 9:  #include "DbOdbcView.h"
10: #include "MoveToDlg.h"
```

B

6. Add a toolbar button for the new menu entry.

Day 15

Quiz

1. What does ADO stand for?

   ActiveX Data Objects.

2. What does ADO use for database access?

   OLE DB.

3. What are the objects in ADO?

   `Connection`, `Command`, `Parameter`, `Error`, `Recordset`, and `Field`.

4. How do you initialize the COM environment?

   `::CoInitialize(NULL);`

5. How do you associate a `Connection` object with a `Command` object?

   `pCmd->ActiveConnection = pConn;`

6. How do you associate a `Command` object with and populate a `Recordset` object?

   One of two ways:

   ```
   _RecordsetPtr pRs;
   pRs = pCmd->Execute();
   ```

   Or

   ```
   _RecordsetPtr pRs;
   pRs.CreateInstance(__uuidof(Recordset));
   pRs->PutRefSource(pCmd);
   ```

Exercise

Enable and disable the navigation menus and toolbar buttons based on whether the recordset is at the beginning of file (`BOF`) or end of file (`EOF`, renamed to `EndOfFile`).

Add event-handler functions to the document class for the navigation menu entries' `UPDATE_COMMAND_UI` event message. Edit these functions, adding the code in Listing B.27 to the functions for the First and Previous menus, and the code in Listing B.28 to the functions for the Last and Next menus.

LISTING B.27. THE `CDbAdoDoc` `OnUpdateDataFirst` FUNCTION.

```
 1: void CDbAdoDoc::OnUpdateDataFirst(CCmdUI* pCmdUI)
 2: {
 3:     // TODO: Add your command update UI handler code here
 4:     // Does the record set exist?
 5:     if (m_pRs)
 6:     {
 7:         // Are we at the BOF?
 8:         if (m_pRs->BOF)
 9:             pCmdUI->Enable(FALSE);
10:         else
11:             pCmdUI->Enable(TRUE);
12:     }
13: }
```

LISTING B.28. THE `CDbAdoDoc` `OnUpdateDataLast` FUNCTION.

```
 1: void CDbAdoDoc::OnUpdateDataLast(CCmdUI* pCmdUI)
 2: {
 3:     // TODO: Add your command update UI handler code here
 4:     // Does the record set exist?
 5:     if (m_pRs)
 6:     {
 7:         // Are we at the EOF?
 8:         if (m_pRs->EndOfFile)
 9:             pCmdUI->Enable(FALSE);
10:         else
11:             pCmdUI->Enable(TRUE);
12:     }
13: }
```

B

Day 16

Quiz

1. When do you want to create a new MFC class?

   When you need to create a new class that is inherited from an existing MFC class.

2. When you make changes to a library file, what do you have to do to the applications that use the library file?

   They all have to be relinked.

3. What are the different types of classes that you can create?

   MFC, generic, and form.

4. When you package some functionality in a library file, what do you need to give to other programmers who want to use your library module?

   The LIB library file and the header files for the objects in the module.

5. What are two of the basic principles in object-oriented software design?

   Encapsulation and inheritance. The third principle is polymorphism, which was not discussed today.

Exercises

Separate the `CLine` class into a different library module from the drawing class so that you have two library modules instead of one. Link them into the test application.

1. Create a new project. Specify that the project is a Win32 Static Library project. Give the project a suitable name, such as `Line`.
2. Specify that the project contain support for MFC and precompiled headers.
3. Copy the `Line.cpp` and `Line.h` files into the project directory. Add both of these files to the project. Compile the library module.
4. Open the original library module project. Delete the `Line.cpp` and `Line.h` files from the project. Edit the include statement at the top of the drawing object source-code file to include the `Line.h` file from the `Line` module project directory, as on line 9 of Listing B.29. Recompile the project.

LISTING B.29. THE `CModArt` INCLUDES AND COLOR TABLE.

```
1:  // ModArt.cpp: implementation of the CModArt class.
2:  //
3:  //////////////////////////////////////////////////////////////////////
4:
```

continues

LISTING B.29. CONTINUED

```
 5:  #include <stdlib.h>
 6:  #include <time.h>
 7:
 8:  #include "stdafx.h"
 9:  #include "..\Line\Line.h"
10: #include "ModArt.h"
```

5. Open the test application project. Add the `Line` library file to the project. Build the project.

Day 17

Quiz

1. What kind of DLL do you have to create to make classes in the DLL available to applications?

   An MFC extension DLL.

2. What do you have to add to the class to export it from a DLL?

   The `AFX_EXT_CLASS` macro in the class declaration.

3. What kind of DLL can be used with other programming languages?

   A regular DLL.

4. If you make changes in a DLL, do you have to recompile the applications that use the DLL?

   Normally, no. Only if changes were made in the exported interface for the DLL do you need to recompile the applications that use the DLL.

5. What function does the LIB file provide for a DLL?

   The LIB file contains stubs of the functions in the DLL, along with the code to locate and pass the function call along to the real function in the DLL.

Exercises

1. Separate the line class into its own MFC extension DLL and use it with the second (regular) DLL.

   Create a new project. Specify that the project is an AppWizard (DLL) project, and give the project a suitable name, such as `LineDll`.

   Specify that the DLL will be an MFC extension DLL.

After generating the project skeleton, copy the line source code and header files into the project directory. Add these files into the project.

Edit the CLine class declaration, adding the AFX_EXT_CLASS macro to the class declaration.

Compile the DLL. Copy the DLL into the debug directory for the test application.

Open the regular DLL project. Delete the line source code and header files from the project in the File View of the workspace pane. Add the line DLL LIB file to the project. Edit the drawing functionality source-code file, changing the line class header include to include the version in the CLine DLL project directory, as in Listing B.30.

LISTING B.30. THE CModArt INCLUDES.

```
 1:  // ModArt.cpp: implementation of the CModArt class.
 2:  //
 3:  //////////////////////////////////////////////////////////////////////
 4:
 5:  #include <stdlib.h>
 6:  #include <time.h>
 7:
 8:  #include "stdafx.h"
 9:  #include "..\LineDll\Line.h"
10: #include "ModArt.h"
```

Compile the project. Copy the DLL into the test application project debug directory.

Run the test application.

2. Alter the line class DLL so that it uses a consistent line width for all lines.

   Open the line class DLL project that you created in the previous exercise. Edit the class constructor, replacing the initialization of the m_nWidth variable with a constant value, as in Listing B.31.

LISTING B.31. THE CLine CONSTRUCTOR.

```
1: CLine::CLine(CPoint ptFrom, CPoint ptTo, UINT nWidth, COLORREF crColor)
2: {
3:     m_ptFrom = ptFrom;
4:     m_ptTo = ptTo;
5:     m_nWidth = 1;
6:     m_crColor = crColor;
7: }
```

Compile the DLL. Copy the DLL into the test application project debug directory. Run the test application.

Day 18

Quiz

1. When is the `OnIdle` function called?

   When the application is idle and there are no messages in the application message queue.

2. How can you cause the `OnIdle` function to be repeatedly called while the application is sitting idle?

   Returning a value of `TRUE` will cause the `OnIdle` function to continue to be called as long as the application remains idle.

3. What is the difference between an `OnIdle` task and a thread?

   An `OnIdle` task executes only when the application is idle and there are no messages in the message queue. A thread executes independently of the rest of the application.

4. What are the four thread synchronization objects?

   Critical sections, mutexes, semaphores, and events.

5. Why shouldn't you specify a higher than normal priority for the threads in your application?

   The rest of the threads and processes running on the computer will receive a greatly reduced amount of processor time.

Exercises

1. If you open a performance monitor on your system while the application that you built today is running, you'll find that even without any of the threads running, the processor usage remains 100 percent, as in Figure 18.11. The `OnIdle` function is continuously being called even when there is nothing to be done.

   Modify the `OnIdle` function so that if there's nothing to be done, neither of the `OnIdle` tasks are active. Then, the `OnIdle` function will not continue to be called until one of these threads is active, at which time it should be continuously called until both threads are once again turned off. This will allow the processor to drop to a minimal utilization, as in Figure 18.12.

   Edit the `OnIdle` function as in Listing B.32.

B

LISTING B.32. THE MODIFIED `CTaskingApp OnIdle` FUNCTION.

```
 1: BOOL CTaskingApp::OnIdle(LONG lCount)
 2: {
 3:     // TODO: Add your specialized code here and/or call the base class
 4:
 5:     // Call the ancestor's idle processing
 6:     BOOL bRtn = CWinApp::OnIdle(lCount);
 7:
 8:     // Get the position of the first document template
 9:     POSITION pos = GetFirstDocTemplatePosition();
10:     // Do we have a valid template position?
11:     if (pos)
12:     {
13:         // Get a pointer to the document template
14:         CDocTemplate* pDocTemp = GetNextDocTemplate(pos);
15:         // Do we have a valid pointer?
16:         if (pDocTemp)
17:         {
18:             // Get the position of the first document
19:             POSITION dPos = pDocTemp->GetFirstDocPosition();
20:             // Do we have a valid document position?
21:             if (dPos)
22:             {
23:                 // Get a pointer to the document
24:                 CTaskingDoc* pDocWnd =
25:                     (CTaskingDoc*)pDocTemp->GetNextDoc(dPos);
26:                 // Do we have a valid pointer?
27:                 if (pDocWnd)
28:                 {
29:                     // Get the position of the view
30:                     POSITION vPos = pDocWnd->GetFirstViewPosition();
31:                     // Do we have a valid view position?
32:                     if (vPos)
33:                     {
34:                         // Get a pointer to the view
35:                         CTaskingView* pView =
               ➥(CTaskingView*)pDocWnd->GetNextView(vPos);
36:                         // Do we have a valid pointer?
37:                         if (pView)
38:                         {
39:                             // Should we spin the first idle thread?
40:                             if (pView->m_bOnIdle1)
41:                             {
42:                                 // Spin the first idle thread
43:                                 pDocWnd->DoSpin(0);
44:                                 bRtn = TRUE;
45:                             }
46:                             // Should we spin the second idle thread?
47:                             if (pView->m_bOnIdle2)
```

continues

LISTING B.32. CONTINUED

```
48:                             {
49:                                 // Spin the second idle thread
50:                                 pDocWnd->DoSpin(2);
51:                                 bRtn = TRUE;
52:                             }
53:                         }
54:                     }
55:                 }
56:             }
57:         }
58:     }
59:     return bRtn;
60: }
```

2. When starting the independent threads, give one of the threads a priority of `THREAD_PRIORITY_NORMAL` and the other a priority of `THREAD_PRIORITY_LOWEST`. Edit the `SuspendSpinner` function as in Listing B.33.

LISTING B.33. THE MODIFIED `CTaskingDoc SuspendSpinner` FUNCTION.

```
 1: void CTaskingDoc::SuspendSpinner(int nIndex, BOOL bSuspend)
 2: {
 3:     // if suspending the thread
 4:     if (!bSuspend)
 5:     {
 6:         // Is the pointer for the thread valid?
 7:         if (m_pSpinThread[nIndex])
 8:         {
 9:             // Get the handle for the thread
10:             HANDLE hThread = m_pSpinThread[nIndex]->m_hThread;
11:             // Wait for the thread to die
12:             ::WaitForSingleObject (hThread, INFINITE);
13:         }
14:     }
15:     else    // We are running the thread
16:     {
17:         int iSpnr;
18:         int iPriority;
19:         // Which spinner to use?
20:         switch (nIndex)
21:         {
22:         case 0:
23:             iSpnr = 1;
24:             iPriority = THREAD_PRIORITY_NORMAL;
25:             break;
26:         case 1:
27:             iSpnr = 3;
28:             iPriority = THREAD_PRIORITY_LOWEST;
```

```
29:             break;
30:         }
31:         // Start the thread, passing a pointer to the spinner
32:         m_pSpinThread[nIndex] = AfxBeginThread(ThreadFunc,
33:             (LPVOID)&m_cSpin[iSpnr], iPriority);
34:     }
35: }
```


Day 19

Quiz

1. What are the three aspects of a control that are visible to the container application?

   Properties, methods, and events.
2. Why do you need to design a property page for your control?

   To provide the user with the ability to set the properties of the control.
3. What are the four types of properties that a control might have?

   Ambient, extended, stock, and custom.
4. What happens to the parameters that are passed to the methods of a control?

   They are marshaled into a standardized, machine-independent structure.
5. What tool can you use to test your controls?

   The ActiveX Control Test Container.

Exercises

1. Add a method to your control to enable the container application to trigger the generation of a new squiggle drawing.

   Open the Class Wizard to the Automation tab. Click the Add Method button. Enter a method name, such as `GenNewDrawing`, and specify the return type as void. Click OK to add the method. Edit the method, adding the code in Listing B.34.

LISTING B.34. THE `CSquiggleCtrl GenNewDrawing` FUNCTION.

```
1: void CSquiggleCtrl:: GenNewDrawing()
2: {
3:     // TODO: Add your specialized code here and/or call the base class
4:     // Set the flag so a new drawing will be generated
5:     m_bGenNewDrawing = TRUE;
6:     // Invalidate the control to trigger the OnDraw function
7:    Invalidate();
8: }
```

2. Add a method to your control to save a squiggle drawing. Use the `CFile::modeWrite` and `CArchive::store` flags when creating the `CFile` and `CArchive` objects.

   Open the Class Wizard to the Automation tab. Click the Add Method button. Enter a method name, such as `SaveDrawing`, and specify the return type as `BOOL`. Add a single parameter, `sFileName`, with a type of `LPCTSTR`. Click OK to add the method. Edit the method, adding the code in Listing B.35.

LISTING B.35. THE `CSquiggleCtrl` `SaveDrawing` FUNCTION.

```
 1: BOOL CSquiggleCtrl::SaveDrawing(LPCTSTR sFileName)
 2: {
 3:     // TODO: Add your dispatch handler code here
 4:     try
 5:     {
 6:         // Create a CFile object
 7:         CFile lFile(sFileName, CFile::modeWrite);
 8:         // Create a CArchive object to store the file
 9:         CArchive lArchive(&lFile, CArchive::store);
10:        // Store the file
11:        m_maDrawing.Serialize(lArchive);
12:    }
13:    catch (CFileException err)
14:    {
15:        return FALSE;
16:    }
17:    return TRUE;
18: }
```

Day 20

Quiz

1. What are the two things that a client application must know to be able to connect to a server application?

   The network address (or name) of the computer and the port on which the server is listening.

2. What `CAsyncSocket` function is used to enable a server application to detect connection efforts by client applications?

   `Listen`.

3. What `CAsyncSocket` member function is called to signal that data has arrived through a socket connection?

   `OnReceive`.

4. What function is called to signal that a connection has been established?

   `OnConnect`.

5. What function do you use to send a message through a socket connection to the application on the other end?

   `Send`.

Exercises

The server application that you wrote can handle only a single connection at a time. If a second application tries to open a connection to it while it has an existing connection to an application, the server application will crash. The server tries to accept the second connection into the socket that is already connected to the first client application. Add a third socket object to the application that will be used to reject additional client connections until the first client closes the connection.

Follow these steps:

1. Add a member variable to the dialog class (`CSockDlg`). Specify the variable type as `BOOL`, the name as `m_bConnected`, and the access as private.
2. Initialize the variable as `FALSE` in the `OnInitDialog` function.
3. Set the variable to `TRUE` in the `OnAccept` dialog function once the connection has been accepted.
4. Set the variable to `FALSE` in the `OnClose` dialog function.
5. Modify the `OnAccept` dialog function as in Listing B.36.

LISTING B.36. THE MODIFIED `CSockDlg` `OnAccept` FUNCTION.

```
 1: void CSockDlg::OnAccept()
 2: {
 3:     if (m_bConnected)
 4:     {
 5:         // Create a rejection socket
 6:         CAsyncSocket sRjctSock;
 7:         // Create a message to send
 8:         CString strMsg = "Too many connections, try again later.";
 9:         // Accept using the rejection socket
10:         m_sListenSocket.Accept(sRjctSock);
11:         // Send the rejection message
12:         sRjctSock.Send(LPCTSTR(strMsg), strMsg.GetLength());
```

continues

LISTING B.36. CONTINUED

```
13:         // Close the socket
14:         sRjctSock.Close();
15:     }
16:     else
17:     {
18:         // Accept the connection request
19:         m_sListenSocket.Accept(m_sConnectSocket);\
20:         // Mark the socket as connected
21:         m_bConnected = TRUE;
22:         // Enable the text and message controls
23:         GetDlgItem(IDC_EMSG)->EnableWindow(TRUE);
24:         GetDlgItem(IDC_BSEND)->EnableWindow(TRUE);
25:         GetDlgItem(IDC_STATICMSG)->EnableWindow(TRUE);
26:     }
27: }
```

Day 21

Quiz

1. What does the `CHtmlView` class encapsulate for use in Visual C++ applications?

   The Internet Explorer Web browser.

2. How can you get the URL for the current Web page from the `CHtmlView` class?

   `GetLocationURL()`.

3. What command is triggered for the frame class when the user presses the Enter key in the edit box on the dialog bar?

   `IDOK`.

4. What functions can you call to navigate the browser to the previous and the next Web pages?

   `GoBack()` and `GoForward()`.

5. How can you stop a download in progress?

   With the `Stop()` function.

Exercises

1. Add the `GoSearch` function to the menu and toolbar.

   Add a menu entry to the Go menu. Specify the menu entry properties in Table B.6.

TABLE B.6. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	`ID`	`IDM_GO_SEARCH`
	`Caption`	`&Search`
	`Prompt`	`Search the Web\nSearch`

Using the Class Wizard, add an event-handler function to the view class on the `IDM_GO_SEARCH` ID for the `COMMAND` event message. Edit the code as in Listing B.37.

LISTING B.37. THE `CWebBrowseView` `OnGoSearch` FUNCTION.

```
1: void CWebBrowseView::OnGoSearch()
2: {
3:     // TODO: Add your command handler code here
4:
5:     // Go to the search page
6:     GoSearch();
7: }
```

Add a toolbar button for the menu ID `IDM_GO_SEARCH`.

2. Add the `GoHome` function to the menu and toolbar.

   Add a menu entry to the Go menu. Specify the menu entry properties in Table B.7.

TABLE B.7. MENU PROPERTY SETTINGS.

Object	*Property*	*Setting*
Menu Entry	`ID`	`IDM_GO_START`
	`Caption`	`S&tart Page`
	`Prompt`	`Go to the start page\nHome`

Using the Class Wizard, add an event-handler function to the view class on the IDM_`GO_START ID` for the `COMMAND` event message. Edit the code as in Listing B.38.

LISTING B.38. THE `CWebBrowseView` `OnGoStart` FUNCTION.

```
1: void CWebBrowseView::OnGoStart()
2: {
3:     // TODO: Add your command handler code here
```

continues

LISTING B.38. CONTINUED

```
4:
5:     // Go to the start page
6:     GoHome();
7: }
```

Add a toolbar button for the menu ID `IDM_GO_START`.

3. Disable the Stop toolbar button and menu entry when the application is not downloading a Web page.

   Using the Class Wizard, add an event handler to the view class for the `IDM_VIEW_STOP` object ID on the `UPDATE_COMMAND_UI` event message. Edit the function, adding the code in Listing B.39.

LISTING B.39. THE `CWebBrowseView` `OnUpdateViewStop` FUNCTION.

```
1: void CWebBrowseView::OnUpdateViewStop(CCmdUI* pCmdUI)
2: {
3:     // TODO: Add your command update UI handler code here
4:
5:     // Enable the button if busy
6:     pCmdUI->Enable(GetBusy());
7: }
```

APPENDIX C

Printing and Print Previewing

by Jon Bates

Using the Framework's Functionality

The SDI and MDI frameworks created by the AppWizard add the hooks for printing and previewing by default. These can be turned off by unchecking the Printing and Print Preview option in Step 4 of the MFC AppWizard, but generally they are useful to include in any project and add very little overhead. Most of the real work of printing is taken care of by the device context and GDI. The framework presents you with a device context for a print document page; you can treat it pretty much as if it's a normal window device context.

Using Default Print Functionality

The SDI (Single Document Interface) framework supports printing images from views based on information held in the document. Because this information is already displayed in your applications views, you can probably print it by modifying the view to add printing support.

The framework calls your `OnDraw()` function in the view to display an image. There is a corresponding `OnPrint()` function that it calls to let your view handle printing the information. Often this task is simply a case of using the same drawing code as you've implemented in your `OnDraw()` function. If this is so, you don't actually need to implement the `OnPrint()` function; the framework does this by default in the `CView` base class and calls `OnDraw()`. The printer is then treated just like it would be for a screen because it offers a device context for the drawing functions to use, as a substitute for the usual screen device context. Your `OnDraw()` function can determine whether the device context it is passed is a screen or printer device context, but because the drawing functions will work in the same way on both, even this knowledge isn't necessary.

You can explore the printing functionality added by the standard framework by creating a standard SDI application with the AppWizard. Leave the Printing and Print Preview option in Step 4 checked (this means you can click Finish on Step 1) and name the project `PrintIt`.

> **Standard Print Framework Support**
>
> The standard print and print preview support is available only in SDI and MDI applications. Dialog box-based applications must implement their own printing support.

The first thing you'll need is a graphic to print. You can create a graphical test display in the `OnDraw()` function of my `CPrintItView` class (just a normal `CView`) as shown in Listing C.1. This test displays a line-art style picture with some centralized text in a large font (see Figure C.1). The test image isn't too important, but it will make a useful comparison between printed output and screen display.

Listing C.1. LST23_1.CPP—Drawing in `OnDraw` to produce a print sample.

```
1:  void CPrintItView::OnDraw(CDC* pDC)
2:  {
3:      CPrintItDoc* pDoc = GetDocument();
4:      ASSERT_VALID(pDoc);
5:
```

```
6:      // TODO: add draw code for native data here
7:
8:      // ** Set metric mapping
9:      pDC->SetMapMode(MM_LOMETRIC);
10:
11:      // ** Declare and create a font 2.2cm high
12:      CFont fnBig;
13:      fnBig.CreateFont(220,0,0,0,FW_HEAVY,FALSE,FALSE,0,
14:          ANSI_CHARSET,OUT_DEFAULT_PRECIS,
15:          CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
16:          FF_SWISS+VARIABLE_PITCH,"Arial");
17:
18:      //** Select the new font and store the original
19:      CFont* pOldFont = pDC->SelectObject(&fnBig);
20:
21:      //** Declare a client rectangle
22:      CRect rcClient;
23:      GetClientRect(&rcClient);
24:
25:      // ** Convert to logical units
26:      pDC->DPtoLP(&rcClient);
27:
28:      // ** Set up some drawing variables
29:      const int nPoints = 50;
30:      int xm = rcClient.Width();
31:      int ym = rcClient.Height();
32:      double dAspW = xm/(double)nPoints;
33:      double dAspH = ym/(double)nPoints;
34:
35:      // ** Select a black pen
36:      CPen* pOldPen =
37:           (CPen*)pDC->SelectStockObject(BLACK_PEN);
38:
39:      // ** Draw the lines
40:      for(int i=0;i<nPoints;i++)
41:      {
42:          int xo = (int)(i * dAspW);
43:          int yo = (int)(i * dAspH);
44:
45:          pDC->MoveTo(xo,0);
46:          pDC->LineTo(xm,yo);
47:          pDC->LineTo(xm-xo,ym);
48:          pDC->LineTo(0,ym-yo);
49:          pDC->LineTo(xo,0);
50:      }
51:
52:      // ** Reselect the old pen
53:      pDC->SelectObject(pOldPen);
54:
```

continues

C

LISTING C.1. CONTINUED

```
55:        // ** Draw the text on top
56:        pDC->SetTextAlign(TA_CENTER+TA_BASELINE);
57:        pDC->SetBkMode(TRANSPARENT);
58:
59:        // ** Set gray text
60:        pDC->SetTextColor(RGB(64,64,64));
61:        pDC->TextOut(xm/2,ym/2,"Sample Print");
62:
63:        // ** Reselect the old font
64:        pDC->SelectObject(pOldFont);
65:    }
```

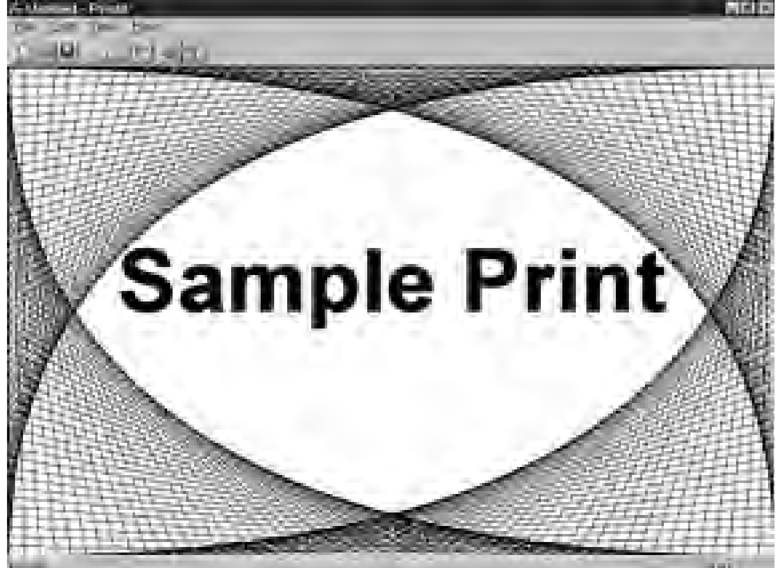


FIGURE C.1.
Graphical test output of PrintIt in a window.

Although there is a fair bit of code in this `OnDraw()` function, none of it is unusual. It just draws lines inside the client rectangle and writes some text in the middle. Notice at line 9, the mapping mode is set to `MM_LOMETRIC`; this sets the logical coordinates to tenths of a millimeter.

A 2.2cm high font is created at line 13 and used to draw the sample text at line 61. Lines 40 to 50 draw the arty “peg and string” frame using the client rectangle coordinates. I’ll let you decipher the details; the important thing here is to investigate the business of printing.

If you build and run the program after adding these lines to the `OnDraw()` function of Listing C.1, you should see a graphical display in your application window, as shown in Figure C.1.

So the big question is this: What must you do to print this image output? Surprisingly little—because the standard framework tries to print this by calling your `OnDraw()` function and passing the device context for the printer rather than for the window.

If you click the File menu of the PrintIt application and choose Print Preview, you'll see a small representation of the image in the top-left corner, although the font is too big for the line drawing. This isn't the framework's fault; it has done its best to represent your window, but it was passed the wrong coordinates for the device context. The problem lies with the `GetClientRect()`used in line 23.

Notice that `GetClientRect()` is a member of the view, not of the device context. This works fine for the window because the device context is the same size as the window rectangle. Now you're passing the window rectangle to the printer device context (which is small in comparison) but creating a 2.2cm high font that is always the same size (because of the mapping mode).

Overriding `OnPrint()`

To fix the client rectangle coordinate size problem, you must pass the correct rectangle for the printer rather than the window. Fortunately, the framework calls a virtual function that you can override in your view and use to find all the information you need. As you read earlier, this function is named `OnPrint()` and is analogous to `OnDraw()`. When drawing in a window, `OnDraw()` is called; when drawing on a printer, `OnPrint()` is called. You might be wondering how the drawing code in `OnDraw()` was executed to print preview the sample graphical display. The default `CView` implementation of `OnPrint()` simply calls `OnDraw()`, passing its printer device context.

Your `OnPrint()` doesn't have to call `OnDraw()`; you can override `OnPrint()` to make it draw something entirely different, but many applications must print out what the user sees. These applications reuse their `OnDraw()` code with the printer device context.

To override the `OnPrint()` virtual function, perform the following steps:

1. Click the ClassView tab of the Project Workspace view.
2. Click the top plus sign to open the view of the project classes.
3. Right-click the view class to which you want to add the `OnPrint()` override (such as `CPrintItView` in the `PrintIt` example) to display the context menu.
4. Select the Add Virtual Function option to display the New Virtual Override dialog box.

5. You should see an `OnPrint` virtual function in the New Virtual Functions list.
6. Click the Add and Edit button to start editing the `OnPrint()` virtual function.

The standard override for `OnPrint()` looks like this:

```
void CPrintItView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here
    CView::OnPrint(pDC, pInfo);
}
```

The first thing you'll notice that's different from `OnDraw()` is the second parameter, the pointer to a `CPrintInfo` object `pInfo`. This is where you'll find the details about the current print, specifically the rectangle coordinates for the printer device context you require. There are lots of useful `CPrintInfo` member variables. Some of these are shown in Table C.1.

TABLE C.1. `CPrintInfo` MEMBER VARIABLES SPECIFIC TO PRINT INFORMATION.

Variable Name	*Description of Contents*
`m_nCurPage`	The current page number for multipage prints
`m_nNumPreviewPages`	Either 1 or 2, depending on the preview pages shown
`m_rectDraw`	The coordinates of the print page rectangle
`m_pPD`	Pointer to a `CPrintDialog` class if the Print dialog box is used
`m_bDirect`	`TRUE` if the Print dialog box has been bypassed
`m_bPreview`	`TRUE` if currently in print preview
`m_strPageDesc`	A format string to help generate the page number
`m_lpUserData`	A pointer that can be used to hold user data

Some other member variables in `CPrintInfo` are covered later in this chapter, but first you'll need to find the printing rectangle coordinates rather than the window's rectangle. The `m_rectDraw` member holds the coordinate rectangle of the current print page. You can use these coordinates with the printer device context in the `OnDraw()` function. There is a problem though, in that this structure isn't passed to the `OnDraw()`, but you can copy the coordinates into a member variable held in your `CPrintItView` class.

Add the following lines to store the rectangle after the `// TODO` comment, but before the `CView::OnPrint()` call:

```
// ** copy the print rectangle from the pInfo
    if (pInfo) m_rcPrintRect = pInfo->m_rectDraw;
```

This will store the printing rectangle in the `m_rcPrintRect` member of the `CPrintItView` class. You must therefore declare this member variable, which is easily done by right-clicking the `CPrintItView` class in the ClassView pane of the Project Workspace view and choosing the Add Member Variable option. The Variable Type is a `CRect`, and the declaration is obviously `m_rcPrintRect`. Access should be private because you don't need or want any other classes to know about this internal rectangle.

Using the Printer Device Context

The device context passed to `OnPrint()` differs slightly from the display context in that it may have fewer colors and is probably larger that your display. Other than these attributes, you can use it to draw in exactly the same way as the screen device context. This is how you can use the same `OnDraw()` to print as well as view in a window. The base class call `CView::OnPrint()` implements code that does exactly this.

The device context holds a flag that you can interrogate via the `IsPrinting()` function to determine whether you are drawing to a screen-based device context or a printer-based device context. You might use this difference to change the printed output from the screen output, or more subtly to adjust the coordinates used to produce the printed output.

For the sample program it only remains to use the `m_rcPrintRect` coordinates when printing in the `OnDraw()` function. The code necessary to use the `IsPrinting()` function to determine whether the window's client rectangle or the printer's rectangle should be used is shown in Listing C.2. The output produced is shown by the print preview in Figure C.2.

LISTING C.2. LST23_2.CPP—ADDING PRINTING RECTANGLE SUPPORT TO THE STANDARD `OnDraw()` IMPLEMENTATION.

```
1:  // Declare a client rectangle
2:  CRect rcClient;
3:
4:  // ** Check the device context for printing mode
5:  if (pDC->IsPrinting())
6:  {
7:      // ** Printing, so use the print rectangle
8:      rcClient = m_rcPrintRect;
9:  }
10:  else
11:  {
12:      // ** Not printing, so client rect will do
13:      GetClientRect(&rcClient);
```

continues

LISTING C.2. CONTINUED

```
14:  }
15:
16:  // Convert to logical units
17:  pDC->DPtoLP(&rcClient);
```

Notice in Listing C.2 that an `if` statement is used in line 5 to call the device context's `IsPrinting()` function. This function returns `TRUE` if this is a printer device context (or preview) and `FALSE` for any other device contexts. In the printing case, you can assign the stored print page rectangle to `rcClient`, as shown in line 8. In the normal screen window case you can just use the standard `GetClientRect()` to find the window's rectangle, as shown in line 13.

Because you've used a mapping mode, you must convert both printing and display rectangle coordinates from device units to logical units. This is done by the `DPtoLP()` function in line 17. If you change and add lines 4–14 to your existing `OnDraw()` function and then build and run the application, you should be able to run the print preview as before, with better results (see Figure C.2).

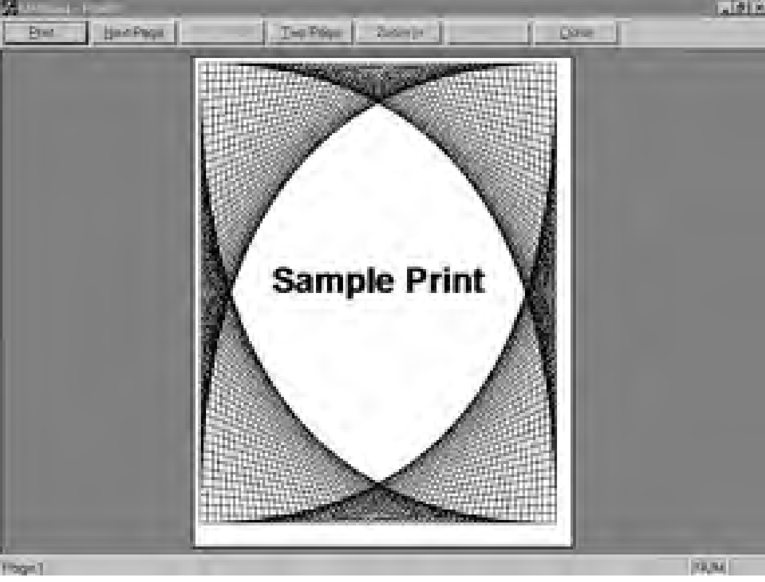


FIGURE C.2.
Print preview using the full print page rectangle coordinates.

Maintaining the Aspect Ratio

As you can see from Figure C.2, because the paper is much longer and thinner than the window, the printed output becomes stretched. The relationship between the width and the height is the aspect ratio. To stop the image from stretching one way or another, you must keep the same aspect ratio as the image in the window. The code in Listing C.2 doesn't try to maintain aspect ratios, which isn't very satisfactory in most cases, so you would need to add some code to maintain the aspect ratio of the printed output.

The best tactic to use in this case is to find out whether setting either the width or the height to the full width or height of the paper will give maximum coverage and then shorten the other dimension to maintain the aspect ratio.

To do this, you need some information about the paper dimensions and its own aspect ratio. There is a device context function that retrieves these details (and many more) named `GetDeviceCaps()`. By passing the `ASPECTX` or `ASPECTY` flags to `GetDeviceCaps()`, you can find the relationship between the width of a pixel and its height. If the relationship is 1:1 the pixel is square; otherwise, it is oblong and might differ from the screen's own aspect ratio. If it differs, you can decide which axis will give you the largest image, while maintaining the same aspect ratio as the screen. That way you can avoid a stretched looking image.

Code that does just that in the `OnDraw()` function is demonstrated in Listing C.3.

C

> **Device Aspect Ratios**
>
> For most printers, you'll probably find that the aspect ratio is 1:1. But if you look closely at thermal printer output like those in fax machines, you can see a very distinctive aspect ratio difference in their pixels.

Listing C.3. LST23_3.CPP—Maintaining the aspect ratio while producing the largest printed representation.

```
1:  //** Declare a client rectangle and get the client rect
2:  CRect rcClient;
3:  GetClientRect(&rcClient);
4:
5:  // ** Check the device context for printing mode
6:  if (pDC->IsPrinting())
7:  {
8:    // ** Find the Print width : Window width ratio
```

continues

LISTING C.3. CONTINUED

```
9:    double dWidthRatio=(double)m_rcPrintRect.Width()/
10:                            (double)rcClient.Width();
11:
12:     // ** Find the Print height : Window height ratio
13:     double dHeightRatio=(double)m_rcPrintRect.Height()/
14:                            (double)rcClient.Height();
15:
16:     // ** Calculate the device's aspect ratio
17:     double dAspect=(double)pDC->GetDeviceCaps(ASPECTX)/
18:                   (double)pDC->GetDeviceCaps(ASPECTY);
19:
20:     // ** Find the new relative height
21:     int nHeight=(int)(rcClient.Height() *
22:                     dWidthRatio * dAspect );
23:
24:     // ** Find the new relative width
25:     int nWidth=(int)(rcClient.Width() *
26:                     dHeightRatio * (1.0 / dAspect) );
27:
28:     // ** Set the whole rectangle
29:     rcClient=m_rcPrintRect;
30:
31:     // ** Determine the best fit across or down the page
32:     if (nHeight > nWidth)
33:     {
34:         // ** Down is best, so adjust the width
35:         rcClient.BottomRight().x=
36:             m_rcPrintRect.TopLeft().x + nWidth;
37:     }
38:     else
39:     {
40:         // ** Across is best, so adjust the height
41:         rcClient.BottomRight().y=
42:             m_rcPrintRect.TopLeft().y + nHeight;
43:     }
44:  }
45:
46:  // Convert to logical units
47:  pDC->DPtoLP(&rcClient);
```

Notice that both the screen window and printed case use the window coordinates that are found from the `GetClientRect()` in line 3. In the onscreen window case, nothing else is done and the code continues as normal.

However, a lot now happens when printing, if the `IsPrinting()` test in line 6 returns `TRUE`. First, you must find the ratios of the window width to the paper width and the

window height to the paper height. You can find these ratios as shown in lines 9 and 13 by dividing the paper dimensions by the window dimensions.

The next thing you must calculate is the device's own aspect ratio peculiarities. You can use the `GetDeviceCaps()` function in line 17 to find the ratio of width to height in the device itself and store the result in `dAspect`.

Using these values, you can now calculate the device's comparative width and height coordinates in terms of the opposing window dimension, as shown in lines 21 and 25. This calculation, which includes the device aspect ratio for each dimension, will yield the adjusted height for the full page width or vice versa. Now you must decide whether you can best fit the full width or height of a page and adjust the other dimension. The condition on line 32 makes this decision based on the bigger width or height. This means that if you have a tall, thin window, it is better to use the full height of the paper and adjust the width; conversely, if you have a short, wide window, it is better to use the full width and adjust the height. Depending on what is better, the adjustment is made on line 35 or 42 by setting the bottom-right point's x- or y-coordinate to the adjusted width or height.

Notice that all the other dimensions are set to `rcClient` from the paper in the assignment on line 29, so the adjustment is the only change required. After this section, the program continues and will use the adjusted rectangle to do its drawing.

If you build and run the application after adding the lines in Listing C.3 to the `OnDraw()` function, you should see that printing or previewing the window will now maintain the same aspect ratio as the onscreen window. If you stretch the window to make it higher than it is wide, the printed output will use the full height of the page rather than the full width, but still maintain the correct aspect ratios.

Pagination and Orientation

Printing a single page to represent the view in a window is a common requirement, but largely the printing process is concerned with printing large and complex multipage documents from the user's sophisticated data. The framework comes to the rescue again and simplifies this process by providing a common Print Setup dialog box and a page enumeration system to print and preview the specified range of pages.

Setting the Start and End Pages

The first considerations for a multipage document are the start and end pages, which also indicate how many pages you are going to print. A framework virtual function in the view class is called first when printing begins. This function is `OnPreparePrinting()`

and it supplies one parameter, the `CPrintInfo` object `pInfo`. This is the first time you'll see the `CPrintInfo`, and this is where you can first change it to customize the print to your requirements. The `OnPreparePrinting()` function is supplied automatically from the AppWizard when you create the SDI, so you don't have to add it yourself. You can see the default implementation by double-clicking the `OnPreparePrinting()` member of the `CPrintItView` class in the ClassView pane.

It should look like this:

```
BOOL CPrintItView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}
```

By default, the `DoPreparePrinting()` function is called and passed the `pInfo` pointer to the `CPrintInfo` object for the print. `DoPreparePrinting()` sets up the required device context and calls the standard Print dialog box if you are printing (not previewing). This dialog box is covered in more detail in the next section, but first you can set up a range of pages to print by modifying the `CPrintInfo` object before the `DoPreparePrinting()`.

To do this, add the following lines before the `// default preparation` comment:

```
pInfo->SetMinPage(2);
    pInfo->SetMaxPage(8);
```

These two member functions of the `CPrintInfo` class will modify the `CPrintInfo` object pointed at by `pInfo` to set the starting page at page 2 via `SetMinPage()` and the ending page at page 8 via `SetMaxPage()`.

Now when the document is printed, the `OnPrint()` function will be called six times. The only difference between each of these calls will be the `pInfo->m_nCurPage` member variable that will hold the current page as it iterates between 2 and 8.

Depending on the kind of application you write, the technique you'll use to determine the number of pages will vary. If you are selling music compact discs and want to print a brochure of your product range, you would probably fit the cover picture and review of each CD on one printed page, so if you sell 120 different CDs, you need 120 pages. However, if you are printing a complex government tender with different bill elements and formatted items, you'll probably need to measure the height of all the different parts and calculate a page count after performing your own pagination. Either way, when you have the page count, `OnPreparePrinting()` is where you'll set it into the `CPrintInfo` object.

Bypassing the Print Dialog Box When Printing

You don't always need to bother the user with the Print dialog box; this can be bypassed by setting the `pInfo->m_bDirect` variable to `TRUE` in `OnPreparePrinting()`.

To emphasize the difference between a full report and a window print, you can implement a completely different drawing in the `OnPrint()` function than `OnDraw()`, as shown in Listing C.4. In this `OnPrint()`, the base class `CView::OnPrint()` function isn't called at all, which means that the default call of `OnDraw()` isn't performed. So in this implementation, the printing output and the display output are entirely different.

Listing C.4. LST23_4.CPP—Implementing page-specific drawing in `OnPrint()`.

```
1: void CPrintItView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
2: {
3:    // TODO: Add your specialized code here
4:
5:   // ** Create and select the font
6:   CFont fnTimes;
7:   fnTimes.CreatePointFont(720,"Times New Roman",pDC);
8:    CFont* pOldFont=(CFont*)pDC->SelectObject(&fnTimes);
9:
10:    // ** Create and select the brush
11:    CBrush brHatch(HS_CROSS,RGB(64,64,64));
12:    CBrush* pOldBrush =
13:        (CBrush*)pDC->SelectObject(&brHatch);
14:
15:    // ** Create the page text
16:    CString strDocText;
17:    strDocText.Format("Page Number %d",
18:                          pInfo->m_nCurPage);
19:
20:    pDC->SetTextAlign(TA_CENTER+TA_BASELINE);
21:
22:    // ** Set up some useful point objects
23:    CPoint ptCenter=pInfo->m_rectDraw.CenterPoint();
24:    CPoint ptTopLeft=pInfo->m_rectDraw.TopLeft();
25:  CPoint ptBotRight=pInfo->m_rectDraw.BottomRight();
26:
27:    // ** Create the points for the diamond
28:    CPoint ptPolyArray[4]=
29:    {
30:        CPoint(ptTopLeft.x,ptCenter.y),
31:        CPoint(ptCenter.x,ptTopLeft.y),
```

continues

LISTING C.4. CONTINUED

```
32:         CPoint(ptBotRight.x,ptCenter.y),
33:         CPoint(ptCenter.x,ptBotRight.y)
34:     };
35:
36:     // ** Draw the diamond
37:     pDC->Polygon(ptPolyArray,4);
38:
39:     // ** Draw the text
40:     pDC->TextOut(ptCenter.x,ptCenter.y,strDocText);
41:
42:     // ** Unselect the fonts
43:     pDC->SelectObject(pOldFont);
44:     pDC->SelectObject(pOldBrush);
45:}
```

In lines 6–12 of Listing C.4, the resources for the print (a font and a brush) are set up. Note that there is a better place to do this, as explained later in this chapter in the section "Adding GDI Objects with `OnBeginPrinting()`."

You can use the current page number to draw the different textual content of each page by its position in the printed document, as shown in line 17. In a real application you would probably use this page number to reference the document and look up a specific item of data. In the compact disc scenario mentioned earlier, this page number might be used to reference a specific CD, and the drawing functions would then use that data. I don't have space to demonstrate anything quite so sophisticated here, so I've just used the current page number from `pInfo->m_nCurPage` to illustrate the point.

Lines 22–37 set up a diamond-shaped polygon to draw as the background and line 40 draws the text containing the current page in the middle of the page. Lines 43–44 reselect the old font and brush.

If you build and run the program after making these changes to `OnPrint()` and then click the test application File menu and choose Print Preview, you should be able to preview multiple pages using the Next Page and Prev Page buttons shown in Figure C.3. If you have a printer attached, you'll also be able to print the multipage document.

Using the Print Dialog Box

Notice that when you print a multipage document, you are first presented with a dialog box that enables you to customize the print settings, as shown in Figure C.4. This is the standard Print dialog box and is called from the `CView::DoPreparePrinting()` function that was called from within the `OnPreparePrinting()` override. This dialog box lets you set the page ranges to print, the number of copies, collation flags, the destination printer, and a whole host of things specific to the printer properties.

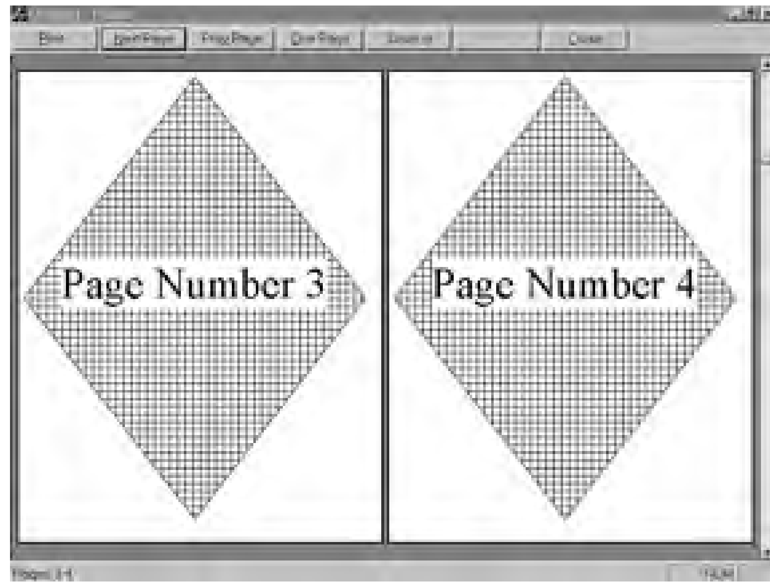


FIGURE C.3.
The Print Preview output of a multipage document.

FIGURE C.4.
The standard Print dialog box.

THE Collate CHECK BOX

If the user unchecks the Collate check box on the Print dialog box, the printer driver will automatically repeat the same pages together. You don't need to do anything special in your code to handle this—but the feature must be supported by the printer driver; otherwise it will be disabled and inaccessible in the Print dialog box.

The user can change the print options from this dialog box, which will then update the settings in the `CPrintInfo` object before it is passed to your application. You can customize this dialog box to a small or great degree depending on the amount of customization you require and the work you're prepared to put into the job.

From the `CPrintInfo` class members in Table C.1, recall that there is an `m_pPD` pointer. This points to a `CPrintDialog` class that is an MFC wrapper class for the Print dialog box. This class also holds an `m_pd` member, which is a `PRINTDLG` structure holding the default settings that are displayed in the Print dialog box. There are many members of this structure, as shown in Listing C.5. This allows complete customization of the dialog box defaults, even to the level of specifying a completely different dialog box template than the default template (if you want a challenge). There isn't enough space here to describe all these members in detail; one of the more obvious members is the `nCopies` member variable. You could change the default number of copies displayed in this dialog box by setting the `nCopies` member of this structure directly before calling the `CView::DoPreparePrinting()` function. To do this, add the following line to your `OnPreparePrinting()` function:

```
pInfo->m_pPD->m_pd.nCopies = 15;
```

When you open the Print dialog box after adding this line, the number of copies will default to 15 (if your printer or printer driver supports multiple copies). You can set the other default values in the `PRINTDLG` accordingly.

> **USING THE `DevMode` STRUCTURE**
>
> The `DevMode` structure holds many useful attributes that describe the technical capabilities and configuration of the device. The structure pointer is returned by the `GetDevMode()` function in the `CPrintDialog` class.

LISTING C.5. LST23_5.CPP—THE `PRINTDLG` STRUCTURE.

```
1:  typedef struct tagPD {
2:  DWORD     lStructSize;
3:  HWND      hwndOwner;
4:  HANDLE    hDevMode;
5:  HANDLE    hDevNames;
6:  HDC       hDC;
7:  DWORD     Flags;
8:  WORD      nFromPage;
9:  WORD      nToPage;
10:  WORD      nMinPage;
11:  WORD      nMaxPage;
12:  WORD      nCopies;
```

```
13:  HINSTANCE hInstance;
14:  DWORD     lCustData;
15:  LPPRINTHOOKPROC lpfnPrintHook;
16:  LPSETUPHOOKPROC lpfnSetupHook;
17:  LPCTSTR    lpPrintTemplateName;
18:  LPCTSTR    lpSetupTemplateName;
19:  HANDLE    hPrintTemplate;
20:  HANDLE    hSetupTemplate;
21:  } PRINTDLG;
```

After the user has confirmed OK in the Print dialog box, you can retrieve the changes by using the `CPrintDialog` class access functions shown in Table C.2. So if you wanted to find the number of copies specified by the user before printing, you could catch the value after it is returned from the `CView::DoPreparePrinting()` function, as shown in Listing C.6.

Obviously, any of the values in the `PRINTDLG` structure `pInfo->m_pPD->m_pd` can be tested here also.

C

TABLE C.2. `CPrintDialog` ACCESS FUNCTIONS.

Function Name	*Description*
`GetCopies()`	Returns the number of copies set by the user
`GetFromPage()`	Returns the starting page as specified
`GetToPage()`	Returns the last page as specified
`GetPortName()`	Returns the selected printer port, for example, LPT1:
`GetDriverName()`	Returns the selected print driver (destination printer)
`GetPrinterDC()`	Returns a device context for the printer
`PrintAll()`	Returns `TRUE` if all pages are selected
`PrintCollate()`	Returns `TRUE` if collation is required
`PrintRange()`	Returns `TRUE` if a range is specified
`PrintSelection()`	Returns `TRUE` if a specific selection of pages is chosen

LISTING C.6. LST23_6.CPP—VALIDATING THE STANDARD PRINT DIALOG BOX FOR A SPECIFIC NUMBER OF COPIES.

```
1:  BOOL CPrintItView::OnPreparePrinting(CPrintInfo* pInfo)
2:  {
3:      pInfo->SetMinPage(1);
4:      pInfo->SetMaxPage(10);
```

continues

LISTING C.6. CONTINUED

```
5:
6:      pInfo->m_pPD->m_pd.nCopies = 3;
7:
8:      do
9:      {
10:          // ** Check if user has cancelled print
11:          if (DoPreparePrinting(pInfo) == FALSE)
12:              return FALSE;
13:
14:          // ** Warn the user if too many copies
            ➥are specified
15:          if (pInfo->m_pPD->GetCopies()>5)
16:              AfxMessageBox("Please choose less than
                ➥5 copies");
17:
18:          // ** Keep looping until they specify a
             ➥valid number
19:       } while(pInfo->m_pPD->GetCopies()>5);
20:       return TRUE;
21:  }
```

In Listing C.6 the `CView::DoPreparePrinting()` returns `FALSE` if the user has pressed Cancel in lines 11 and 12. Otherwise, the number of copies set is checked in line 15, and a warning is issued if more than five copies have been selected (my arbitrary criteria). The loop is repeated at line 19 until the user enters a valid number of copies or presses Cancel.

Using Portrait and Landscape Orientations

If you click the File menu from the application and choose the Print Setup option, you can change the printer's orientation defaults. You can choose either Portrait or Landscape from the dialog. You don't need to make any code changes to handle Landscape printing; if you choose this option and then run a print preview, you should notice that the device context is now drawn to the shape of the paper turned on its side. As long as your application takes note of the `rectDraw` member of the `CPrintInfo` object, it should be able to cope with landscape printing automatically.

Adding GDI Objects with `OnBeginPrinting()`

As I mentioned earlier, the code in Listing C.4 works fine, but there is a better way to allocate the resources needed. Currently every time a page is printed, `OnPrint()` is called to draw the page, and all the resources are created from scratch. That probably won't slow things down too much for this simple output, but in a large, complex report you

might want to set up a number of resources and other calculations just once at the start of the report. Then you can print a number of pages and clean up the resources at the end of the report.

The `OnBeginPrinting()` virtual function is an ideal place to do this initialization, and its sister function, `OnEndPrinting()`, is the place to clean up these resources. `OnBeginPrinting()` is called after `OnPreparePrinting()` and is the first place where a printer device context is passed in. This device context is the one that is used during the printing process, so you can set up all the GDI objects and printer page coordinates at this point. The default code supplied automatically by the ClassWizard just gives you an empty function:

```
void CPrintItView::OnBeginPrinting(CDC* /*pDC*/,
➥ CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}
```

Take a close look at that function definition. Notice the parameters are actually commented out to the compiler, throwing warning messages about unused parameters when you compile. You'll have to remember to uncomment these parameters before you start using them.

You can now add the GDI object creation calls to this function to avoid doing it on every page:

```
m_fnTimes.CreatePointFont(720,"Times New Roman",pDC);
m_brHatch.CreateHatchBrush(HS_CROSS,RGB(64,64,64));
```

Notice that the `fnTimes` and `brHatch` objects have been prefixed by an `m_`; this is a naming convention to indicate that the objects have class scope (are embedded in the class) rather than local scope (are embedded in the function). Because you'll need to access these GDI objects in `OnPrint()`, you can add them to the class declaration. You can do this by adding the font and brush objects to the class declaration like this:

```
protected:
    CFont     m_fnTimes;
    CBrush    m_brHatch;
```

You can add these either by double-clicking the `CPrintItView` class in the `ClassView` and adding them directly or by using the Add Member Variable dialog box.

Also notice that the hatched brush is created with the `CreateHatchBrush()` function rather than with the constructor. This is because the brush will exist as long as the view does, but you must call `DeleteObject()` in the `OnBeginPrinting()` function so that the underlying GDI resource is freed between prints. You can add the code to delete both the font and brush GDI objects in `OnEndPrinting()`, as shown in these lines:

```
m_fnTimes.DeleteObject();
m_brHatch.DeleteObject();
```

All that remains is to remove the local GDI objects from the `OnPrint()` function itself and replace their references with the member variable versions. You can do this by replacing the `CFont fnTimes` and `CBrush brHatch` local variables and their creation functions and just selecting the precreated font and brush:

```
CFont* pOldFont = (CFont*)pDC->SelectObject(&m_fnTimes);
CBrush* pOldBrush = (CBrush*)pDC->SelectObject(&m_brHatch);
```

If you were to build and run the application after making these changes, you'd probably notice no difference. Functionally it's the same, but the print and preview should be a little faster. If you had a large, complex 100-page report using lots of GDI resources, you'd definitely find this technique useful in speeding up the printing.

USING COORDINATES FROM `OnBeginPrinting()`

You might be tempted to also store the coordinates from `OnBeginPrinting()`. This won't work because `CPrintInfo`'s `m_rectDraw` member hasn't been initialized by that stage and random coordinates will be used.

Customizing Device Context Preparation

Before both `OnDraw()` and `OnPrint()` are called, the `OnPrepareDC()` virtual function is called and can be overridden in your view class to perform any device context modifications that might be common to both `OnDraw()` and `OnPrint()`. You might want to set mapping modes or set certain common draw modes to the device context for both onscreen and printing modes. The override isn't supplied by the AppWizard, but can easily be added from the Add Virtual Function dialog box. One thing common to both `OnDraw()` and `OnPrint()` in the example is the `SetTextAlign()` device context function. You could add this to an `OnPrepareDC()` function like this:

```
void CPrintItView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    pDC->SetTextAlign(TA_CENTER+TA_BASELINE);
}
```

There might be times, especially when preparing WYSIWYG printouts, that it is advantageous to set mapping modes and window extents in a common function before the draw or print function is called. `OnPrepareDC()` is the place to put any device context–specific initialization code.

Aborting the Print Job

Another use of `OnPrepareDC()` is to call printer escapes or other print document–specific functions. If you had a particularly long report, you might want to give the user the option of terminating the printing process and aborting the print. The `AbortDoc()` device context function aborts the printing document for a printer device context. You can try this by adding the following lines to `OnPrepareDC()` and aborting the document after three pages:

```
if (pDC->IsPrinting())
    if (pInfo->m_nCurPage==3) pDC->AbortDoc();
```

Direct Printing Without the Framework

So far in this chapter, I've shown you the SDI and MDI framework support for printing. This support melds nicely into the Document/View architecture, but there are times when you just want quick and easy access to a printer or don't have the framework available—in a dialog-based application, for example.

The framework support hides lower-level printing support that is the bedrock for all the printing operations. This section explains how this support works and shows it in use in a dialog box-based application example.

Invoking the Print Dialog Box Directly

You saw in the earlier section "Using the Print Dialog Box" how the `CPrintDialog` class provides a wrapper for the common `PRINTDLG` dialog and how this was called from `CView::DoPreparePrinting()`.

The same dialog box and class can be used directly to set up the destination printer and its default settings just like you'd use a normal modal dialog box. You can use the same access functions to set the page numbers and copy defaults as you used from inside the framework's `DoPreparePrinting()` function.

Listing C.7 shows this dialog box being used directly to configure the printer for dialog box-based printing and then prints a small document from the defaults set by the dialog box.

The direct printing mechanism works via the `StartDoc()` and `EndDoc()` functions shown in this listing and is explained in the next section.

You can use the AppWizard to create a dialog box-based application named `DlgPrint` and create an `OnOK()` handler with the ClassWizard to implement the printing code, as shown in Listing C.7.

LISITNG C.7. LST23_7.CPP—IMPLEMENTING A DIRECT DOCUMENT PRINT IN OnOK OF A DIALOG BOX-BASED APPLICATION.

```
 1:  void CDlgPrintDlg::OnOK()
 2:  {
 3:      // TODO: Add extra validation here
 4:
 5:      // ** Construct a CPrintDialog object
 6:      CPrintDialog dlgPrint(FALSE,PD_ALLPAGES,this);
 7:
 8:      if (dlgPrint.DoModal()==IDOK)
 9:      {
10:          // ** Attach the printer DC from the dialog
11:          // ** to a CDC object
12:          CDC dcPrint;
13:          dcPrint.Attach(dlgPrint.GetPrinterDC());
14:
15:          // ** Create and fill a DOCINFO structure
16:          DOCINFO myPrintJob;
17:          myPrintJob.cbSize = sizeof(myPrintJob);
18:          myPrintJob.lpszDocName = "MyPrintJob";
19:          myPrintJob.lpszOutput = NULL;
20:          myPrintJob.lpszDatatype = NULL;
21:          myPrintJob.fwType = NULL;
22:
23:          // ** Start the printing document
24:          if (dcPrint.StartDoc(&myPrintJob)>=0)
25:          {
26:              // ** Start a page
27:              dcPrint.StartPage();
28:
29:              // ** Start drawing
30:              dcPrint.TextOut(0,0,"My Small Print Job");
31:
32:              // ** Throw the page
33:              dcPrint.EndPage();
34:
35:              // ** Close the document
36:              dcPrint.EndDoc();
37:          }
38:
39:          // ** Delete the printer device context
40:          dcPrint.DeleteDC();
41:       }
42:
43:       // ** Carry on with the standard OnOK
44:       CDialog::OnOK();
45:   }
```

Listing C.7 declares a `CPrintDialog` object `dlgPrint` at line 6 that takes three parameters in its constructor. The first parameter is a flag that can be set as `TRUE` to display the Print Setup dialog box, or `FALSE` to display the Print dialog box. The second parameter is a set of combinable flags that customize the settings of the dialog box (too numerous to cover here). The third parameter is a pointer to the parent window; in this case the C++ `this` pointer indicates that the dialog box is the parent.

On line 8, `dlgPrint.DoModal()`is called to display this dialog box. If the user clicks OK, the print begins; otherwise, the block is skipped.

When the user has clicked OK in the Print dialog box, a device context for the printer is created and attached to a `CDC` object in line 13 to make it easier to use. You must remember to delete the device context itself, as shown in line 40.

You can add the listing lines and handler, build and run it, and click OK of the dialog box application to run the new code.

Using `StartDoc()` and `EndDoc()`

The `CDC` device context has many printer-specific functions. To start a new print, Windows must create a spool document to store the print job and submit it to the printer when it is complete. The `StartDoc()` function tells Windows to start spooling, and the `EndDoc()` function tells it that the document is complete and can be sent to the printer. You saw the `AbortDoc()` function earlier that will abort the print and cancel the print job rather than send to the printer.

Listing C.7 calls the `StartDoc()` member of the printer device context object `dcPrint` at line 24, passing a pointer to a `DOCINFO` structure. This structure holds the details of the print job. The only detail you must specify is a name for the spool document, which is assigned at line 18. Notice that it has an unusual `cbSize` member that holds the size of the structure. This is assigned the value from `sizeof(myPrintJob)` at line 17. You see this sort of strange action going on a lot at the Win32 API level because `DOCINFO` is an old C-style structure; the `cbSize` is used because there are a few different forms of `DOCINFO` and the only way to tell them apart is the size.

When `StartDoc()` is called, it will try to start the print job and return a positive value if it succeeds. There are many reasons why it might fail, such as low disk space or memory, or a corrupt printer driver, so it’s a good idea to carry on with the print only after checking the return code.

After the document is printed, you should call `EndDoc()` as shown on line 36 to start printing the document.

WATCHING THE WINDOWS SPOOLER

You can watch the print document as it builds up by placing a breakpoint in the `OnPrint()` function or after a `StartDoc()` function and opening your printer status icon from the Printers group available from the main Windows Start menu under the Settings option.

Using `StartPage()` and `EndPage()`

Another pair of printer device context functions are `StartPage()` and `EndPage()`. The `StartPage()` function is used to initialize the device context ready for printing a new page. This will reset some of the device context settings such as the current graphics cursor position and set the document spooling information for starting a new page.

Typically, you'd call `StartPage()`, do some drawing in the device context for the details to be printed on that page, and call `EndPage()` to write the page away to the spool file to add it to the print document.

In Listing C.7, `StartPage()` is called on line 27, followed by a solitary `TextOut()` function to draw something on the printer page, followed by a call to `EndPage()` on line 33.

When `EndPage()` is called, the special printer codes for throwing a `Form Feed` are sent to the spooler and the spool document registers another print page. You can repeat this `StartPage()` and `EndPage()` sequence for all the document pages before calling `EndDoc()` to complete the printing process. You can use the printer device context for drawing in just the same way as the `OnPrint()` was used in the SDI application in between the `StartPage()` and `EndPage()` calls. The same functions were called in the SDI framework, but the framework hides it from you, only calling your `OnPrint()` between start and end page calls.

APPENDIX D

Understanding and Exception Handling

by Jon Bates

Using Exceptions

An *exception* is an object that holds details about something that has gone wrong. The clever thing about exception handling is that you can create an exception when something goes wrong in a low-level function and have it automatically bubble back up to a calling function that can deal with all such exceptions in one place.

Running Code and Catching the Errors

The system automatically detects certain error conditions and generates exceptions for them. If you don't deal with them in your application, they will bubble back out of your code and be handled by Windows's own exception-catching mechanisms. If you want to see this in action, just add the following two lines to any of your code and run it:

```
CDC* pNullDC = 0;
pNullDC->SetPixel(0,0,0);
```

The first line declares a device context pointer `pNullDC` and sets it to point the memory address to zero (which isn't a good place for any object to be). Obviously there isn't a valid object at this address, so when the following `SetPixel()` function is called, the system tries to find the object at address zero. The memory management hardware and software know that the program has no right being at this memory address and raise a memory access violation exception.

If you run these lines of code from outside the Visual C++ debugger, you'll see a dialog box familiar to all Windows users, as shown in Figure D.1.

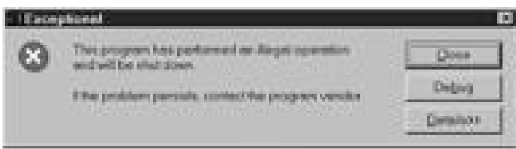


FIGURE D.1.
The Windows memory access violation Exception dialog box.

However, if you run the application from the Visual C++ debugger, the debugger first catches the exception for you and displays the Developer Studio dialog box instead, as shown in Figure D.2.

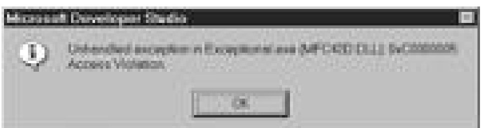


FIGURE D.2.
The Developer Studio-handled memory access violation exception.

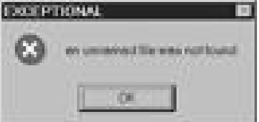
Memory access violations are very severe exceptions that will crash your program without any chance to catch them. There are many less severe exceptions, such as the file-handling exception `CFileException`. This exception is thrown when erroneous file operations occur, such as attempting to seek to the beginning of an unopened file:

```
CFile fileNotOpen;
fileNotOpen.SeekToBegin();
```

This results in a system-generated dialog box (see Figure D.3). If you click OK, your program will continue as usual.

FIGURE D.3.

The Windows default dialog box for a file exception.



Rather than letting the system catch the exception, you can catch the exception and deal with it yourself in a more graceful manner. To do this, you must use the C++ `try` and `catch` keywords. You can use these by defining a block of code to `try`; then, when a specified exception is raised, an action is defined in the `catch` block (see Listing D.1).

LISTING D.1. LST29_1.CPP—USING A `try` AND `catch` BLOCK TO CATCH `CFileExceptions`.

```
1: // ** Try a block of code
2: try
3: {
4:     CFile fileNotOpen;
5:     fileNotOpen.SeekToBegin();
6: }
7: catch(CFileException* e)    // Catch File Exceptions
8: {
9:     // ** Check the cause of the exception
10:     if (e->m_cause == CFileException::fileNotFound)
11:        AfxMessageBox("Oops, forgot to open the file!");
12:     e->Delete();
13: }
```

D

In Listing D.1, a `try` block is defined around the file operations at lines 4 and 5. If these lines don't raise an exception, the code will continue as normal. However, if a `CFileException` is raised, it will be caught by the `catch` keyword in line 7, and the variable `e` will point to the new exception. The `CFileException` object has an `m_cause` code that defines exactly why the exception was raised. This is checked on line 10, and if this was a `CFileException::fileNotFound` code, the message box on line 11 is displayed.

Notice that the `Delete()` member function of the `CException` class (the base class of `CFileException`) in line 12 will delete the exception for you. You must ensure that exceptions are always deleted when you are finished with them.

The `try` block can include calls to other functions and could be used to catch any specified exceptions raised in a large portion of the application, as shown in Listing D.2.

LISTING D.2. LST29_2.CPP—A `try` BLOCK CAN INCLUDE MANY FUNCTION CALLS AND CALLS FROM THOSE FUNCTIONS.

```
1: try
2: {
3:     // ... Lots of code
4:     DoLotsOfFileHandling();
5:     // ... More code
6:     EvenMoreFileHandling();
7:     // ... And more Code
8: }
9: catch(CFileException* e)    // Catch File Exceptions
10: {
11:     // ** Check the cause of the exception
12:     if (e->m_cause == CFileException::fileNotFound)
13:        AfxMessageBox("Oops, forgot to open the file!");
14:     e->Delete();
15: }
```

In Listing D.2 the `DoLotsOfFileHandling()` function on line 4 could implement some file handling itself, as well as calls to other functions, as with `EvenMoreFileHandling()` on line 6. Should a file exception arise through any of these file operations, the exception will bubble back so that the same `catch` block will be executed in lines 9 through 13 with `e` pointing to the `CFileException` object. Finally the exception is deleted in line 14.

If you want to catch two different exceptions from the `try` block, you can add `catch` blocks to handle each different exception, as shown in Listing D.3.

LISTING D.3. LST29_3.CPP—HANDLING TWO DIFFERENT EXCEPTIONS WITH THE EXCEPTION-SPECIFIC `catch` BLOCKS.

```
1: try
2: {
3:     // ** This file operation is ok
4:     CMemFile fileMemFile;
5:     fileMemFile.SeekToBegin();
6:
7:     // ** But you can't have two different system
8:     // ** resources with the same name.
9:     CMutex        mutex1(0,"Same Name");
10:     CSemaphore    semaphore1(1,1,"Same Name");
11: }
12: catch(CFileException* e)    // Catch File Exceptions
13: {
14:     if (e->m_cause == CFileException::fileNotFound)
15:        AfxMessageBox("Oops, forgot to open the file!");
16:     e->Delete();
```

```
17: }
18: catch(CResourceException* e)
    ➥// Catch Resource Exceptions
19: {
20:     // ** Report the Resource exception error
21:     AfxMessageBox("Oops, duplicate resource name");
22:     e->Delete();
23: }
```

In Listing D.3, the memory file is automatically created in line 4, so line 5 won't cause a file exception. However, naming two different system resources (a mutex and a semaphore) with the same name does cause a `CResourceException` in line 10 that is then caught by the second `catch` block in line 18, which displays the message box in line 21. If you try this code yourself, remember to add an `#include <afxmt.h>` line for the `CMutex` and `CSemaphore` definitions.

If you want to do a blanket exception catch, you don't need to have a `catch` block for each type of exception; instead, you can catch the `CException` base class exception from which all the other more specific exception classes are derived (see Listing D.4).

LISTING D.4. LST29_4.CPP—USING THE `catch` BLOCK TO CATCH ALL TYPES OF EXCEPTIONS.

```
1: // ** Try a block of code
2: try
3: {
4:     // ** Lots of code ...
5: }
6: catch(CException* e)
7: {
8:     // ** General Error message, details in e
9:     AfxMessageBox("Oops, something went wrong!");
10:     e->Delete();
11: }
```

Notice that on line 6 the `CException` base class is used rather than a specific exception such as `CFileException` or `CResourceException`. You can test which type of exception was raised using the `IsKindOf()` function inside the `catch` block. For example, to test whether a file exception has been raised, you might use the following lines:

```
if (e->IsKindOf(RUNTIME_CLASS(CFileException)))
AfxMessageBox("File Exception");
```

Because exceptions are derived from `CObject`, they support the MFC runtime class information. By using `DECLARE_DYNAMIC` and `IMPLEMENT_DYNAMIC`, the class information

is bundled into the derived exception object so that the `IsKindOf()` function can be used to check for a specific class type. The `RUNTIME_CLASS` macro turns class names into a pointer to a `CRuntimeClass` object for the specified object. The `IsKindOf()` member function will then return `TRUE` if the object being called is of that runtime class.

The "MFC Exception Types" section later in this chapter covers how you can determine exception-specific information from each type of MFC exception caught in a `catch` block.

> **FREEING SYSTEM RESOURCES**
>
> This exception-catching principle becomes very useful when you want to detect and handle errors arising from large portions of code. It can save coding lots of individual error-checking lines, but you must still free up any system resources that you've allocated in lines before the exception was raised.

Throwing Exceptions

You can throw exceptions yourself from code embedded in any enclosing `try` block when an error condition arises. The corresponding `catch` block will then handle the exception. Or you can throw the exception again from within a `catch` section to a higher-level `catch` section, enclosing the first.

Several `AfxThrow...` functions will automatically generate and throw various types of MFC exceptions up to the next catch level, such as `AfxThrowFileException()` or `AfxThrowMemoryException()`. These are covered in detail in the "MFC Exception Types" section. However, these functions create a new instance of a specific `CException`-derived object for you—using the C++ `new` keyword and then the `throw` keyword to raise an exception, as shown in the code fragment in Listing D.5.

LISTING D.5. LST29_5.CPP—RAISING AN EXCEPTION WITH THE `throw` KEYWORD.

```
1: try
2: {
3:     DoSomeFileHandling();
4: }
5: catch(CFileException* e)
6: {
7:     e->ReportError();
8:     e->Delete();
9: }
10:
```

```
11: return TRUE;
12: }
13:
14: BOOL bSomeThingWentWrong = TRUE;
15:
16: void CExceptionalDlg::DoSomeFileHandling()
17: {
18:     // ** ... File handling functions
19:     if (bSomeThingWentWrong == TRUE)
20:     {
21:         CFileException* pException =
22:            new CFileException(CFileException::generic);
23:         throw(pException);
24:     }
25:
26: // ** ... Yet More file handling
27: }
```

In Listing D.5 the `try` block encloses a call to the `DoSomeFileHandling()` function in line 16. This function may implement some file-handling procedures and raises an exception when the error condition on line 19 is found to be `TRUE`. Line 22 creates a new `CFileException` object passing the `CFileException::generic` flag to its constructor and then `throw`s the new object in line 23 to be caught by the `catch` section in line 5.

This process of newing a `CException`-derived object and then using the `throw` keyword is the basis of the exception-raising mechanism. The specific details indicating the cause of the error can be attached to the `CException` object, or extra information can be added by deriving a class from the `CException` base class and adding extra variables to store more specific information.

Your `catch` block can then determine whether the error is too severe to be handled at that level. If so, you might want to throw the exception out to a higher-level enclosing `catch` block. You can use the `throw` keyword (with no parameters) from within the `catch` block to rethrow the exception before you delete it. Instead of deleting the exception, you could rethrow it to a higher level `catch` block by changing the `catch` block shown in Listing D.5 to add the `throw` keyword like this:

```
e->ReportError();
throw;
```

Then after reporting the error, the exception will be thrown again for an enclosing `try` block to catch. If you haven't implemented this nesting, the overall MFC outside the `catch` block will catch it. You can use this nesting mechanism to determine the error severity and implement appropriate recovery mechanisms at various hierarchical levels in your program.

Deleting Exceptions

As you've seen, you are fundamentally responsible for `new`-ing exceptions and must also delete these objects when you've handled them. If you delete one of the MFC exceptions, you shouldn't use the normal C++ `delete` keyword (as you've seen) because the exception might be a global object or a heap object. Instead, the `CException` base class has a `Delete()` function that first checks to see whether the exception should be deleted. The creator of the exception can specify whether the exception should be deleted or not by passing `TRUE` into the `b_AutoDelete` parameter of the `CException` class's constructor (which is the only parameter).

MFC Exception Types

The Microsoft Foundation Classes have several predefined `CException`-derived classes that are used during different types of MFC operations. You've already seen `CFileException` and `CResourceException` in use. The following section covers each of these various classes and how it is raised in more detail. Each class is based on the `CException` class and extends the functionality of `CException` for different types of exception handling. You can also derive your own exception classes from `CException`, and a generic `CUserException` is used for user-oriented application exceptions.

Using the `CException` Base Class

`CException` itself has a constructor that takes an `AutoDelete` flag as discussed earlier, and is defined like this:

```
CException( BOOL b_AutoDelete );
```

If you new a `CException` or derived class, you should ensure that this is set to `TRUE` so that it will be deleted with the C++ `delete` keyword. Otherwise, a global or stack-based exception should pass `TRUE` so that it is deleted only when it goes out of scope (at the end of a function or program that declares it).

The base class contains the `Delete()`function and two error-reporting functions. `GetErrorMessage()` can be used to store the error message into a predefined buffer and specify the ID of a help message to show the user context-specific help pertinent to the error. Its first parameter is the address of a destination buffer to hold the associated error message. The second parameter specifies the maximum size of the buffer so that messages stored in the buffer don't over-spill outside the buffer area. The third optional parameter can specify the context help ID as a `UINT` value.

You might use this function to help format an error message more relevant to your application:

```
char msg[512];
e->GetErrorMessage(msg,sizeof(msg));
CString strMsg;
strMsg.Format("The following error occurred in
➥MyApp: %s",msg);
AfxMessageBox(strMsg);
```

The `sizeof()` C++ operator in the `GetErrorMessage()` function returns the size of an array or variable, so if the `msg` array is changed, you don't have to change any other code. The message is then formatted into the `strMsg` `CString` object and displayed in a message box.

The `ReportError()`function displays the message text directly in the familiar exception message box and would be used from the `catch` block:

```
e->ReportError();
```

Using the Memory Exception

The `CMemoryException` is raised automatically when a C++ `new` keyword fails. You can also raise it yourself using the `AfxThrowMemoryException();` function. The meaning of this exception is exclusively that Windows can't allocate any more memory via its `GlobalAlloc()` or other memory allocation functions. This is a pretty dire situation for any program; you would usually handle this exception by writing code that lets your program die gracefully, freeing up memory and system resources as it goes. There are rare cases in which you could take recovery action if you had a large block of memory allocated and could free it without too much detriment to the users' activities.

Due to the exclusivity of this exception, no other cause attributes or specific functions extend the `CException` class's functionality.

You can watch `new` automatically raise a `CMemoryException` with these lines:

```
MEMORYSTATUS mem;
GlobalMemoryStatus(&mem);
BYTE* pBig = new BYTE[mem.dwAvailVirtual+1];
```

The `mem.dwAvailVirtual` structure member of `MEMORYSTATUS` will hold the total available memory after the `GlobalMemoryStatus()` function retrieves the details. The `new` on the next line requests one more byte than it could possibly have, thus throwing the exception.

Using the Resource Exceptions

CResourceException is thrown in many places where system resources are compromised, as you saw in the mutex and semaphore example in Listing D.3. If you want to throw these exceptions yourself, use the corresponding AfxThrowResourceException() function.

Windows can't find or allocate the requested resource and doesn't give any more specific guidance; hence it has no other functions or attributes.

Using the File and Archive Exceptions

You already looked at CFileException in Listing D.5. This is probably one of the more sophisticated MFC exceptions because of the number of things that can go wrong with file access. You can throw these yourself using the AfxThrowFileException() function, which takes three parameters, one mandatory and the other two optional. The first mandatory parameter, cause, is a cause code for the exception. This will be placed in the file exception's m_cause member variable for interrogation in a catch block.

Table D.1 shows a list of the various cause codes. The second parameter, lOsError, can be used to specify an operating system error code to be placed in the file exception's m_lOsError member variable. This long value can help clarify an error in more detail by drawing on the operating system's own list of file access errors. The third parameter, strFileName, is placed into the file exception's m_strFileName member string variable to indicate the filename of the file that was being accessed when the error occurred.

TABLE D.1. THE CFileException m_cause CODES.

Cause Code	*Meaning*
CFileException::none	There was no error.
CFileException::generic	No error code specified.
CFileException::tooManyOpenFiles	Too many concurrently open files.
CFileException::fileNotFound	Can't find the specified file.
CFileException::badPath	The path name specified is invalid.
CFileException::invalidFile	An attempt was made to use an invalid file handle.
CFileException::badSeek	The seek operation failed.
CFileException::endOfFile	The end of the file was reached.
CFileException::diskFull	There is no spare disk space.
CFileException::hardIO	A hardware error occurred.
CFileException::accessDenied	Permissions deny access to the file.

Cause Code	*Meaning*
`CFileException::directoryFull`	The directory has too many files and can't add another.
`CFileException::removeCurrentDir`	Can't remove the current working directory.
`CFileException::lockViolation`	Can't lock an already locked region of the file.
`CFileException::sharingViolation`	A shared region is locked or can't be shared.

There is also a `ThrowOsError()`static member function that throws and configures a file exception based on an operating system error code. You must pass `ThrowOsError()` the operating system error code as its first parameter and an optional filename as its second parameter. Another member function, `ThrowErrno()`, does the same thing but uses the UNIX-style `errno` error codes as its only parameter (from the `Errno.h` header file). Because these are static functions, you would use them with static scope to raise exceptions with lines like this:

```
CFileException::ThrowOsError(ERROR_BAD_PATHNAME);
➥// Invalid Path
CFileException::ThrowErrno (ENOSPC); // Disk Full
```

Another static member function, `OsErrorToException()`, automatically converts between operating system error codes and `CFileException` cause codes. By passing an OS error code, it will return the corresponding cause code. A corresponding function `ErrnoToException()` does the same when passed an `errno` error code.

When using archives with the `CArchive` class, you normally handle both `CFileExceptions` and `CArchiveException` cases in conjunction: Many of the `CArchive` operations are tied in with their underlying file and file access functions. `CArchiveException` has its own `m_cause` member to hold archive-specific cause codes, as shown in Table D.2. You can raise archive exceptions yourself through the `AfxThrowArchiveException()` function, which requires a `cause` code parameter and a `lpszArchiveName` string pointer for the archive object throwing the exception.

TABLE D.2. THE `CArchiveException` `m_cause` CODE VALUES.

Cause Code	*Meaning*
`CArchiveException::none`	No error occurred.
`CArchiveException::generic`	The specific cause wasn't specified.
`CArchiveException::badSchema`	The wrong version of an object was read.
`CArchiveException::badClass`	The class of the object being read was unexpected.
`CArchiveException::badIndex`	The file format is invalid.

continues

D

TABLE D.2. CONTINUED

Cause Code	*Meaning*
CArchiveException::readOnly	Attempt to write on an archive opened for loading.
CArchiveException::writeOnly	Attempt to read on an archive opened for storing.
CArchiveException::endOfFile	The end of the file was reached unexpectedly while reading.

Using the Database Exceptions

There are two database exception classes: CDBException is used for ODBC-based database access, and CDAOException is used for DAO-based database access. You can throw these exceptions yourself with the AfxThrowDBException() function, which needs three parameters. The first, nRetCode, specifies one of a huge number of database return codes to define the type of error (you should look in the ODBC documentation for these). The second parameter, pDB, is a pointer to the database associated with the exception, and the third parameter, hstmt, is an ODBC handle to the SQL statement object that was executed, causing the exception.

The RETCODE type is available from the CDBException object via its m_nRetCode member. You can also access a human-readable piece of error text from the m_strError member string and the error text returned from the ODBC driver itself in the m_strStateNativeOrigin member.

The CDAOException class has a corresponding AfxThrowDaoException() function that can throw the DAO exception objects. This function needs just two optional parameters. The first, nAfxDaoError, is a DAO-specific error code that indicates problems with DAO itself (see Table D.3). The second parameter is an OLE SCODE value that is the return code from a DAO-based OLE call (see the section "Using OLE Exceptions" for a definition of SCODEs).

TABLE D.3. DAO COMPONENT-SPECIFIC ERROR CODES FROM nAfxDaoError.

Error Code	*Meaning*
NO_AFX_DAO_ERROR	The exception was due to a DAO-specific problem; you should check the supplied CDaoErrorInfo object and SCODE value.
AFX_DAO_ERROR_ENGINE_INITIALIZATION	The Microsoft Jet Engine database engine failed during initialization.
AFX_DAO_ERROR_DFX_BIND	A DAO record set field exchange address is invalid.
AFX_DAO_ERROR_OBJECT_NOT_OPEN	The queried table hasn't been opened.

The `CDAOException` class has three member attributes: `m_scode`, which holds an associated OLE `SCODE` value with the attempted operation; or `S_OK`, if the OLE operation was successful. The `m_nAfxDaoError` member holds one of the DAO-specific values from Table D.3. The `m_pErrorInfo` is a pointer to a `CDaoErrorInfo` structure that holds an error code, descriptive error strings, and a help context ID that is defined like this:

```
struct CDaoErrorInfo
{
   long m_lErrorCode;
   CString m_strSource;
   CString m_strDescription;
   CString m_strHelpFile;
   long m_lHelpContext;
};
```

By interrogating this structure, you can find most of the specific database error details pertaining to the DAO exception.

DAO exceptions can describe more than one error at a time, so you can use the `GetErrorCount()` member function to find out how many are being referenced. These other errors can then be obtained by passing the `GetErrorInfo()` function a zero-based index to the specific error. After calling `GetErrorInfo()` with a specific index in the range returned by the `GetErrorCount()` function, `m_pErrorInfo` will be updated to point to the specified object, and thus you can retrieve those values.

Using OLE Exceptions

There are two types of OLE exceptions, represented by two classes: the `COleException` class, which is normally used for server-side or OLE-specific operations, and the `COleDispatchException` class, which is used when dealing with client-side `IDispatch`-based operations such as calling ActiveX object functions.

The simpler of the two is the `COleException` class, which can be generated by calling the `AfxThrowOleException()` function passing an OLE `SCODE` value. An OLE `SCODE` is a 32-bit error code that is used to represent any kind of error arising from an OLE function.

This value would probably arise from the return code of a function call to a function on one of the interfaces of an OLE object. This `SCODE` value will then be stored in the exception's `m_sc` member for analysis from within a `catch` block.

There is also a `Process()` static member function that is passed an exception object and will turn that exception into an `SCODE` value to represent that exception.

The COleDispatchException class is used in conjunction with OLE IDispatch interfaces and is thrown by the AfxThrowOleDispatchException() function. This function has two forms, both with two mandatory parameters and an optional parameter. The first parameter for both forms is a wCode WORD value that is an application-specific error code. The second parameter is an lpszDescription string pointer in one form, or nDescriptionID for a UINT resource code; both types represent either a verbal string or a string resource code for a verbal string describing the error. The last optional parameter is a help context ID.

These values are then available as member variables of the COleDispatchException object via m_wCode, m_strDescription, and m_dwHelpContext. If a help context is specified and a help file available, the framework will fill in an m_strHelpFile string identifying the help file. The name of the application producing the error can also be sought from the m_strSource attribute.

If you raise this exception from an OLE object such as an ActiveX control, Visual Basic or any other application using the control or object will display these exception details.

Using the Not Supported Exception

The CNotSupportedException class represents exception objects that are generated when an unsupported MFC, operating system, or user-application–specific feature is requested. If you want to raise this exception, use AfxThrowNotSupportedException(), which doesn't required any parameters. There are also no extended members or functions associated with this exception—it just means unsupported.

Using the User Exception

You can use the CUserException class to generate application-specific exception objects. You might want to do this when your program is interacting with the user to halt the process should she choose a certain option. For example, when you are using the AppWizard, you can press Esc at any time to cancel the whole process. Microsoft might have used CUserException to do this by detecting the Esc key and then raising a user exception object.

This exception can be raised by a call to the AfxThrowUserException() function and then caught in the usual try and catch blocks. There are some places in the MFC where this exception is raised, such as during dialog box validation or if the file is too big for an edit view.

Generating Your Own Custom Exception Classes

You can derive your own exception classes from `CException` and add your specific extended functionality. Listing D.6 shows the class definition for such a custom exception class that extends the normal functionality by adding a `m_strMessage` `CString` variable to the exception, enabling you to specify your own message when constructing the exception.

LISTING D.6. LST29_6.CPP—CLASS DEFINITION FOR `CCustomException` IMPLEMENTED IN `CustomException.h`.

```
1:  // ** CustomException.h
2:  // ** Header file for CCustomException
3:
4:  class CCustomException : public CException
5:  {
6:      DECLARE_DYNAMIC(CCustomException);
7:
8:  public:
9:      CCustomException(CString strMessage);
10:
11:      CString m_strMessage;
12:  };
```


In Listing D.6 the class is implemented in its own `CustomException.h` header file and derives from `CException` in line 4. The `DECLARE_DYNAMIC` macro in line 6 supplies the MFC `CObject`-derived runtime class information required for you to decide the exception type in a catch-all `catch` block. The constructor definition in line 9 takes a `CString strMessage` parameter to let you create the custom exception with the message that will be stored in the `m_strMessage` `CString` variable declared in line 11.

The corresponding `CCustomException` class implementation is shown in Listing D.7.

LISTING D.7. LST29_7.CPP—IMPLEMENTATION OF THE `CCustomException` CLASS.

```
1:  // ** CustomException.cpp
2:  // ** Implementation for CCustomException exception
3:
4:  #include "stdafx.h"
5:  #include "CustomException.h"
6:
7:  IMPLEMENT_DYNAMIC(CCustomException,CException);
8:
9:  CCustomException::CCustomException(CString strMessage)
10:      : m_strMessage(strMessage)
11:  {
12:  }
```

In Listing D.7 the usual header files are included, and the IMPLEMENT_DYNAMIC macro is used in line 7 to implement the MFC runtime class information functions. The constructor in line 9 takes the strMessage parameters and initializes the m_strMessage member variable with this string value in line 10.

You can then use the custom exception class in your application, as shown in Listing D.8.

LISTING D.8. LST29_8.CPP—USING THE NEW CCustomException CLASS.

```
1: try
2: {
3:     // ** Something goes wrong
4:     CCustomException* pCustomEx =
5:         new CCustomException("My custom error occured");
6:     throw(pCustomEx);
7: }
8: catch(CCustomException* e)
9: {
10:     // ** Access the extended m_strMessage string
11:     AfxMessageBox(e->m_strMessage);
12:     e->Delete();
13: }
```

In Listing D.8 a new CCustomException object is created with the application-specific error text in lines 4 and 5 and is thrown in line 6. This is then caught by the catch keyword in line 8 and the custom information used by the message box in line 11. The exception is then deleted in line 12.

If you try this, remember that the implementation code must also have an #include for the CustomException.h header file to retrieve the class definition like this:

```
#include "CustomException.h"
```

APPENDIX E

Using the Debugger and Profiler

by Jon Bates

Creating Debugging and Browse Information

A large part of application development is actually debugging your program. All software development is a tight cycle of application design, implementation, and debugging.

Visual C++ has an extensive debugging environment and a range of debugging tools that really help with program development. You can quickly identify problems, watch the contents of variables, and follow the flow of programs through your own code and the MFC code.

Tools such as the Spy++ program can show you the messages passed between Windows and your application and let you spy on applications to see which user interface controls and Window styles they use.

Using Debug and Release Modes

There are two main compiler configurations that you can set to build your application: Debug and Release mode. You can change these modes by clicking the Project menu and selecting the Settings option or by pressing Alt+F7, which will display the Project Settings dialog box (see Figure E.1). The main project settings are shown at the top level and can be changed by selecting the options listed in the combo box. When one setting is selected, changes that you make to any options on the tabs on the right will be set against that configuration. When you build the application, it will be built using your current configuration settings, or you can select All Configurations to build and make changes to all configurations simultaneously.

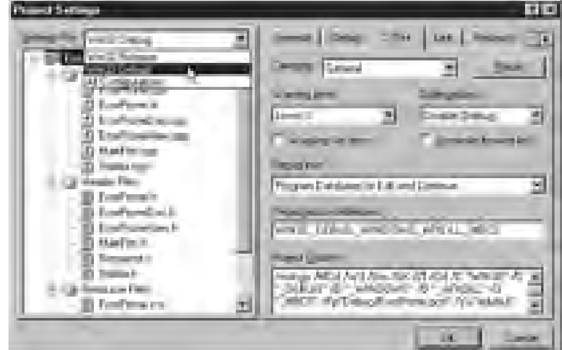


FIGURE E.1.
The C/C++ tab of the Project Settings dialog box.

Both Release and Debug configurations are supplied whenever you create a new project; they produce very different object code. When configured for Debug mode, your build will produce a large and fairly slow executable program. This is because lots of debugging information is included in your program and all the compiler optimizations are disabled.

When you compile the same program in Release mode, you'll see a small, fast executable program, but you won't be able to step through its source code or see any debugging messages from it.

Normally, when developing an application, you leave the compiler set to Debug mode so that you can easily spot and debug problems that arise in your code. When you've finished your application and are preparing to release it, you can set the configuration to Release mode and produce a small, fast program for your users.

> **RELEASE MODE TESTING**
>
> You should always fully test your application after rebuilding it in Release mode before sending it to users. Bugs can arise from things such as leaving proper program code in ASSERT macros (discussed later this chapter), which are then removed, or because of the effect of some speed and memory optimizations.

Setting Debug Options and Levels

You can set a variety of debugging options and levels from the C/C++ tab of the Project Settings dialog box. This dialog page is available from the Project menu by selecting the Settings option (or by pressing Alt+F7) and then selecting the C/C++ tab.

With the General Category selected, the following items are available:

- Warning Level. This is the level of compiler warning messages given during compilation. You can set it to any of the values shown in Table E.1. The default level is Level 3, which is quite sensitive, although many good C++ programmers insist on using Level 4 to get the most warning of potential problems from the compiler. Level 1 and no warnings (None) should be used only in special circumstances because they indicate only severe warnings (or none at all).

> **LEVEL 4 WARNINGS**
>
> At level 4, you'll find that Microsoft's own AppWizard-generated code gives warnings (although usually only about unused function parameters that can be safely ignored).

- Warnings as Errors. When you check this, warning messages are shown as errors that then stop the compiler.
- Generate Browse Info. When you check this, the compiler generates information that can be used to help you locate functions, symbols, and class relationships shown in a Browse window (discussed in the next section). Unfortunately, generating this useful information increases the compilation time quite a bit for large projects (where you most need it).
- Debug Info. This lets you specify the level of debugging information generated by the compiler, as shown in Table E.2.
- Optimizations. In Debug mode, you would normally leave these disabled because they interfere with the debugging process and take longer to compile. However, in

Release mode you can decide whether to Maximize Speed or Minimize Size of your application (or a default that compromises to get the best of both).

- Preprocessor Definitions. This specifies manifest definitions that are defined when your program is compiled. You can use these definitions in conjunction with the `#ifdef`, `#else`, and `#endif` preprocessor commands to compile sections of code in specific configurations. The `_DEBUG` definition is set by default when in Debug mode. You can use this to compile Debug mode–only code in your application like this:

```
int a = b * c / d + e;
#ifdef _DEBUG
CString strMessage;
strMessage.Format("Result of sum was %d",a);
AfxMessageBox(strMessage);
#endif
```

  The message box code is then compiled and run when your application is built in Debug mode. When you switch to Release mode, the code isn't compiled into your executable.

- Project Options. The compiler itself runs as a console-based application and converts your Developer Studio options into several flags to be passed on the command line. You can add additional flag settings for more obscure compiler settings that don't have a user interface switch directly into this edit box.

TABLE E.1. COMPILER WARNING LEVELS.

Level	*Warnings Reported*
None	None
Level 1	Only the most severe
Level 2	Some less severe messages
Level 3	Default level (all reasonable warnings)
Level 4	Very sensitive (good for perfectionists)

TABLE E.2. DEBUG INFO SETTINGS.

Setting	*Debugging Information Generated*
None	Produces no debugging information—usually reserved for Release modes.
Line Numbers Only	This generates only line numbers that refer to the source code for functions and global variables. However, compile time and executable size are reduced.

Setting	*Debugging Information Generated*
C 7.0–Compatible	This generates debugging information that is compatible with Microsoft C 7.0. It places all the debugging information into the executable files and increases their size, but allows full symbolic debugging.
Program Database	This setting produces a file with a `.pdb` extension that holds the maximum level of debugging information, but doesn't create the Edit and Continue information.
Program Database for Edit and Continue	This is the default and usual debug setting. It produces a `.pdb` file with the highest level of debugging and creates the information required for the new Edit and Continue feature.

Creating and Using Browse Information

You can use the Source Browser tool to inspect your source code in detail. This tool can be invaluable if you are examining someone else's code or coming back to your own code after you haven't viewed it for awhile.

To use the Source Browser, you must compile the application with the Generate Browse Info setting checked, in the C/C++ tab of the Project Settings dialog box. To run the tool, press Alt+F12 or click the Tools menu and select the Source Browser option. (The first time you run the tool, it will ask you to compile the browser information.)

The first dialog box the Source Browser presents requests an Identifier to browse for (as shown in Figure E.2). This identifier can be a class name, structure name, function name, or global or local variable name in your application. After you have entered an identifier, the OK button is enabled, and you can browse for details about that identifier.

FIGURE E.2.

The Browse dialog box requesting a symbol to browse.

Select Query offers various options for details pertaining to your chosen symbol. You can choose from any of the following:

- Definitions and References. This option shows you all the files that have references to the specified identifier and whether they are references to the identifier

(places where it is used in the code) or definitions (places where the identifier is defined), as shown in Figure E.3. The line numbers are listed along with the filenames in each file. By double-clicking one of the references or definitions, the code to which it refers will be loaded and shown in the Developer Studio editor at that specific position. This is very useful for tracking all the places that a specific variable or function is used.

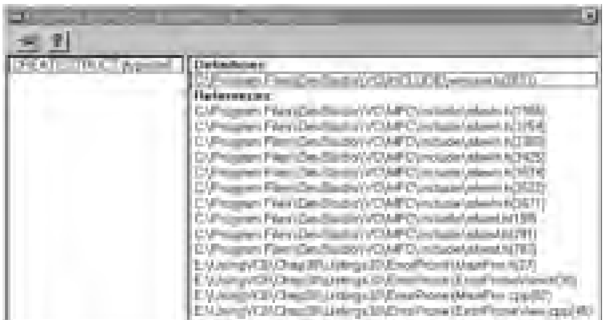


FIGURE E.3.

Source Browser showing definitions and references.

- File Outline. This option shows you all the classes, data, functions, macros, and types that are defined in the specified filename (identifier), as shown in Figure E.4. You can filter each type in or out by pressing relevant buttons along the top of the browser window.

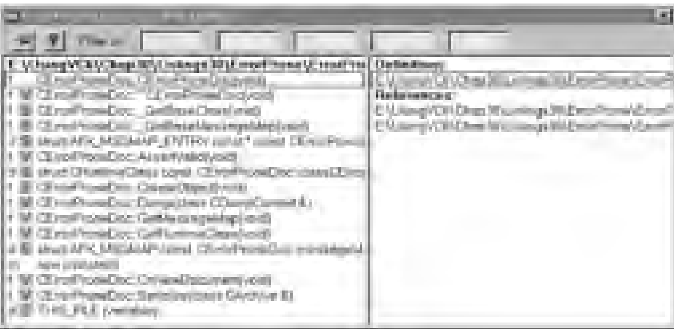
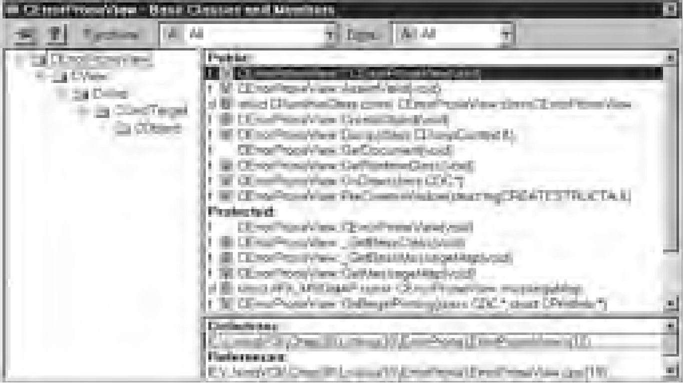


FIGURE E.4.

The file outline display of the source browser.

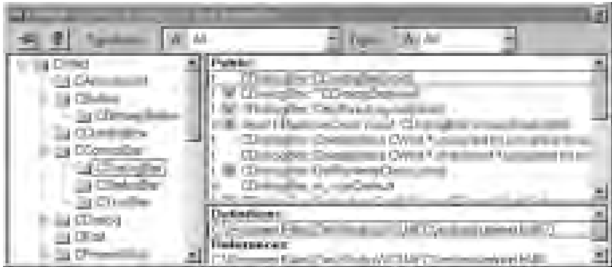
- Base Classes and Members. This arguably is one of the most useful options of the source browser. By specifying a class as the identifier, all the classes' hierarchy and member functions and variables at each hierarchy level are displayed (see Figure E.5). You can also set the filter options to show only certain types of member functions and variables.

FIGURE E.5.
The Base Classes and Members view of the source browser.



- Derived Classes and Members. This view is also very useful and shows all the classes that are derived from the specified class, along with their own member functions and variables. You can also use the browser with the MFC classes to gain more insight into the MFC implementation, as shown with the MFC `CWnd` class in Figure E.6.

FIGURE E.6.
The Derived Classes and Members view of the Source Browser showing `CWnd`-derived classes.



- Call Graph. The Call Graph option shows you all the functions that are called by a specified identifier and the files in which they are defined and implemented. This lets you quickly track the potential flow of a program.
- Callers Graph. The corresponding Callers Graph option shows you all the functions that call the specified identifier. You can use this to track the possible callers of your specified function.

Using Remote and Just-in-Time Debugging

The debugger includes tools that let you debug a program running on a remote machine (even over the Internet via TCP/IP). This can be useful if you want to test

your application in a different environment other than your development machine. To do this, you must have exactly the same versions of the `.dll` and `.exe` files on both machines. After loading the project, you can debug it via a shared directory from the remote machine by changing the Executable for Debug Session edit box to the path and filename of your local `.exe` file (located in the Project Settings dialog box under the Debug tab).

You must also add a path to the `.exe` file in the Remote Executable Path and File Name edit box at the bottom of the Debug tab, leaving the Working Directory blank. You can then start the remote debugger monitor on the remote computer by running the `MSVCMON.EXE` program and connecting to it by clicking the Build menu and selecting the Debugger Remote Connection option.

From the Remote Connection dialog box you can choose Local for a shared directory debug session or Remote to debug via a TCP/IP connection. (You can set the address by clicking Settings.) This will connect to the remote monitor that will start the remote debugging session.

Installing the Remote Debugger Files

You will also need the following files to run the remote debugger monitor on the remote machine: `MSVCMON.EXE`, `MSVCRT.DLL`, `TLN0T.DLL`, `DM.DLL`, `MSVCP50.DLL`, and `MSDIS100.DLL`. These files can be found in your installed `...\Microsoft Visual Studio\Common\MSDev98\bin` subdirectory.

Just-in-time debugging lets you debug a program that was run normally (not through the debugger) and then developed a problem. If you have Visual C++ installed on a machine and this option is enabled, any program that develops a fault will be loaded into a new Developer Studio session ready for debugging and show the code that caused the crash.

This often raises a chuckle when Developer Studio itself crashes and then proceeds to load another session of itself, offering you an assembly code view of where the crash took place in the original for you to debug. It can be very useful to debug your own applications when they crash unexpectedly (usually in a demonstration to your boss). You can enable this option by clicking the Tools menu and selecting Options to display the Options dialog box. Then select the Debug tab and ensure that the Just-in-Time debugging check box is checked.

The OLE RPC debugging option on this tab is also very useful when developing COM and DCOM applications because it lets the debugger traverse a function call into another

out-of-process program or `.dll` and lets another debugger take over for the other process. It then hands control back when returning from the remote function and works across networks and different computers.

Tracing and Single Stepping

One of the most useful features of the Visual C++ debugging environment is the interactive single stepping. This feature lets you step through the code one line at a time and examine the contents of variables as you go. You can also set breakpoints so that the program runs until it reaches a breakpoint and then stops at that point, letting you step from that point until you want to continue running.

Trace statements and assertions are also very useful tools for finding program faults. Trace statements let you display messages and variables from your program in the output window as it runs through trace statements. You can use assertions to cause the program to stop if a condition isn't `TRUE` when you assert that it should be.

Using the `TRACE` Macro

You can add `TRACE` macros to your program at various places to indicate that various parts of the code have been run or to display the contents of variables at those positions. The `TRACE` macros are compiled into your code in the debug configuration and displayed in the Output window on the Debug tab, when you run your program through the debugger.

You can safely leave in the `TRACE` macros when you perform a release build because these macros are automatically excluded from the destination object.

You can display simple messages or output variable contents by passing a format string as the first parameter to the `TRACE` macro. This format string is exactly the same as you would pass to a `printf()` or `CString::Format()` function. You can specify various special formatting codes such as `%d` to display a number in decimal, `%x` to display a number in hexadecimal, or `%s` to display a string. The following parameters should then correspond to the order of the formatting codes. For example, the code

```
int nMyNum = 60;
char* szMyString = "This is my String";
TRACE("Number = %d, or %x in hex and my string is: %s\n",
          nMyNum, szMyString);
```

will result in this output trace line:

```
Number = 60, or 3c in hex and my string is
➥This is my String
```

Listing E.1 shows the TRACE macro used to display the contents of an array before and after sorting by a very inefficient but simple sort algorithm.

If you want to try the code shown in Listing E.1, you can use the AppWizard to build a simple SDI framework. Simply add the code above the OnNewDocument() member function of your document class and then call it by adding a DoSort() call into your OnNewDocument() function.

You can run the application through the debugger (click Build, select Start Debug, and choose Go from the pop-up menu) to see the output trace.

You must ensure that the output window is visible (click the View menu and select Output) when the tabbed output window is shown (same as the compiler output). Ensure that the Debug tab is selected.

LISTING E.1. LSTE_1.CPP—A SIMPLE SORT ROUTINE TO DEMONSTRATE DEBUGGING TECHNIQUES.

```
1:  void Swap(CUIntArray* pdwNumbers,int i)
2:  {
3:      UINT uVal = pdwNumbers->GetAt(i);
4:      pdwNumbers->SetAt(i, pdwNumbers->GetAt(i+1));
5:      pdwNumbers->SetAt(i+1,uVal);
6:  }
7:
8:  void DoSort()
9:  {
10:     CUIntArray arNumbers;
11:     for(int i=0;i<10;i++) arNumbers.Add(1+rand()%100);
12:
13:     TRACE("Before Sort\n");
14:     for(i=0;i<arNumbers.GetSize();i++)
15:         TRACE("[%d] = %d\n",i+1,arNumbers[i]);
16:
17:     BOOL bSorted;
18:     do
19:     {
20:         bSorted = TRUE;
21:         for(i=0;i<arNumbers.GetSize()-1;i++)
22:         {
23:             if (arNumbers[i] > arNumbers[i+1])
24:             {
25:                 Swap(&arNumbers,i);
26:                 bSorted = FALSE;
27:             }
28:         }
29:     } while(!bSorted);
30:
```

```
31:      TRACE("After Sort\n");
32:      for(i=0;i<arNumbers.GetSize();i++)
33:          TRACE("[%d] = %d\n",i+1,arNumbers[i]);
34:  }
```

Listing E.1 sorts an array of random numbers (between 1 and 100), generated in line 11. Lines 13 to 15 then print out the contents of the array before sorting by `TRACE` statements. Lines 17 through 29 sort the array by swapping pairs of numbers that are in the wrong order (by calling the `Swap()` function in line 25). The `Swap()` function (lines 1 to 6) takes a pointer to the array and a position and then swaps the two numbers at that position.

After sorting, the contents of the array are again printed in the output window by the `TRACE` statements in lines 31 to 33.

The trace output of this program appears in the Output window of Developer Studio, as shown in Table E.3.

TABLE E.3. OUTPUT FROM THE SORTING PROGRAM.

Before Sort	*After Sort*
[1] = 42	[1] = 1
[2] = 68	[2] = 25
[3] = 35	[3] = 35
[4] = 1	[4] = 42
[5] = 70	[5] = 59
[6] = 25	[6] = 63
[7] = 79	[7] = 65
[8] = 59	[8] = 68
[9] = 63	[9] = 70
[10] = 65	[10] = 79

Using the ASSERT and VERIFY macros

You can use the `ASSERT` macro to ensure that conditions are `TRUE`. `ASSERT` is passed one parameter that is either a `TRUE` or `FALSE` expression. If the expression is `TRUE`, all is well. If the expression is `FALSE`, your program will stop and the Debug Assertion Failed dialog box will be displayed (see Figure E.7), prompting you to Abort the program, Retry the code, or Ignore the assertion. It also shows the program, source file, and line number

where the assertion failed. If you choose Abort, the debugging session is terminated. Retry is probably the most useful option because the compiler will then show you the code where the ASSERT macro has failed, enabling you to figure out what went wrong. If you already know or don't care about the assertion, you can choose Ignore and continue running the program, which might then result in a more fatal error.

FIGURE E.7.
The Debug Assertion Failed dialog box helps you track down bugs.

A common use of ASSERT is to ensure that input parameters to functions are correct. For example, you can make the Sort() function (shown in Listing E.1) more robust by checking its input parameters. To check the input parameters, add ASSERT macros at the top of the Sort() function like this:

```
ASSERT(pdwNumbers);
ASSERT(i>=0 && i<10);
```

This will ensure that the pointer to the numbers array isn't zero and that the position to swap is between 0 and 9. If either of these is incorrect, the Debug Assertion Failed dialog box is displayed. This technique helps you track down errors caused by passing faulty parameters to functions. It is a good practice to use the ASSERT macro to check that the values passed to each of your functions conform to your expectations.

Another macro, ASSERT_VALID, can be used with CObject-derived classes such as most MFC classes. This performs a more thorough check on the object and its contents to ensure the entire object is in a correct and valid state. You can pass a pointer to the object to be checked like this:

```
ASSERT_VALID(pdwNumbers);
```

Another ASSERT macro is ASSERT_KINDOF, which is used on CObject-derived classes to check that they are of the correct class type. For example, you can check that a pointer to your view object is of the correct view class like this:

```
ASSERT_KINDOF(CYourSpecialView,pYView);
```

The Assertion Failed dialog box will be displayed if it isn't of the correct class type or any of its derivatives.

You must be careful not to put any code that is needed for normal program operation into ASSERT macros because they are excluded in the release build. A common source of release mode errors that are hard to track down is coding like this:

```
int a = 0;
ASSERT(++a > 0);
if (a>0) MyFunc();
```

In the debug build, this code will increment the integer a in the ASSERT line and then call MyFunc() in the following line because a is greater than zero. When your sales team is eager to demonstrate your new application, you might think it works fine because there aren't any Debug mode problems. So you recompile it in Release mode and hand it over to your sales department, which demonstrates it to a customer, whereupon it crashes badly. It crashes because the ++a isn't performed—the release mode excludes ASSERT lines.

The VERIFY macro helps with this problem. VERIFY works like ASSERT, and in Debug mode it throws the same Assertion Failed dialog box if the expression is zero. However, in release mode the expression is still evaluated, but a zero result won't display the Assertion dialog box. You will tend to use VERIFY when you always want to perform an expression and ASSERT when you only want to check while debugging. Therefore, replacing ASSERT in the previous example with VERIFY, as shown in the following example, will enable the release build to work properly:

```
VERIFY(++a > 0);
```


You are more likely to use VERIFY to check return codes from functions:

```
VERIFY(MyFunc() != FALSE);
```

Using Breakpoints and Single Stepping the Program

The use of single stepping and breakpoints is probably the most effective debugging tool for tracking down the majority of problems. The support for various types of breakpoints and the single-stepping information available is very sophisticated in Visual C++; I can only hope to give you a taste of the power of this debugging tool.

The key to single stepping is breakpoints. You can set a breakpoint anywhere in your code and then run your program through the debugger. When the breakpoint is reached, the code will be displayed in the editor window at the breakpoint position, ready for you to single step or continue running.

You can add a breakpoint by selecting the specific code line (clicking the editor cursor onto the line in the editor window) and then either clicking the Breakpoint icon in the Build minibar (see Figure E.8) or by pressing F9. Alternatively, most sophisticated breakpoints can be added or removed by clicking the Edit menu and selecting the Breakpoints option to display the Breakpoints dialog box (see Figure E.9). When you add a breakpoint, it's displayed as a small red circle next to the line you have specified. Breakpoints can be set only against valid code lines, so sometimes the Developer Studio will move one of your breakpoints to the closest valid code line for you.

FIGURE E.8.

Adding a breakpoint to your code via the Build minibar toolbar or the F9 key.

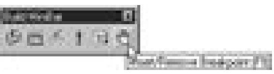
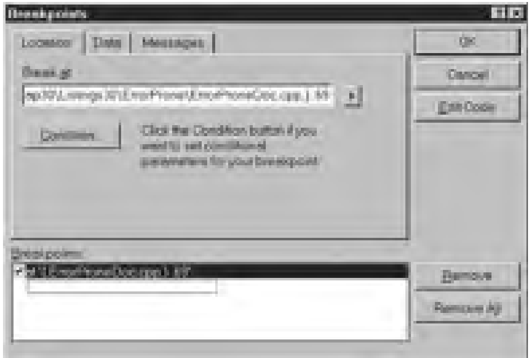


FIGURE E.9.

Adding a breakpoint using the Breakpoints dialog box.



You can toggle the breakpoint on or off by clicking the Breakpoint (hand shaped) icon or remove it by clicking the Remove or Remove All buttons on the Breakpoints dialog box. You can leave them in position but disable them by clicking the check mark to the left of each breakpoint listed in the Breakpoints dialog box. Clicking there again will show the check and re-enable the breakpoint.

When you have set your breakpoint(s), you can run the code through the debugger by choosing Build, Start Debug, Go. Alternatively, you can use the shortcut by clicking the Go icon (to the left of the Breakpoint icon on the Build minibar toolbar—refer to Figure E.8) or by pressing the F5 key.

The program will run as normal until it reaches the breakpoint, where it will stop and display an arrow against the line with the breakpoint. At that point, you can use the Debug toolbar to control the single stepping process, as shown in Figure E.10.

FIGURE E.10.

The debugger stopped at a breakpoint ready for single stepping with the Debug toolbar.

When stopped in the debugger, you can see the contents of most variables merely by moving the cursor over them in the editor window. Their contents are then displayed in a ToolTip at the cursor position. More detailed contents are shown by dragging the variables into the Watch window, as discussed in detail in the next section.

You can single step through the code using the four curly brace icons shown on the Debug toolbar or by clicking the Debug menu and choosing one of the step options. The available step options are shown in Table E.4. You can find these on the Debug menu and the Debug toolbar.

TABLE E.4. STEP OPTIONS AVAILABLE IN SINGLE STEPPING.

Icon/Step Option	*Shortcut Key*	*Effect When Selected*
Step Into	F11	The debugger will execute the current line and if the cursor is over a function call, it will enter that function.
Step Over	F10	Like Step Into except when over a function call line, it will run that function at normal speed and then stop when it returns from the function, giving the effect of stepping over it.
Step Out	Shift+F11	The debugger will run the rest of the current function at normal speed and stop when it returns from the function to the calling function.
Run to Cursor	Ctrl+F10	The debugger will run until it reaches your specified cursor position. You can set this position by clicking the line you want to run to.
Go	F5	Continue running the program at normal speed until the next breakpoint is encountered.
Stop Debugging	Shift+F5	This stops the debugger and returns to editing mode.

continues

TABLE E.4. CONTINUED

Icon/Step Option	*Shortcut Key*	*Effect When Selected*
Restart	Ctrl+Shift+F5	This option restarts the program from the beginning, stopping at the very first line of code.
Break Execution		This option stops a program running at normal speed in its tracks.
Apply Code Changes	Alt+F10	This option lets you compile the code after making changes during a debugging session and then continue debugging from where you left off.

By using these options, you can watch the flow of your program and see the contents of the variables as they are manipulated by the code. The yellow arrow in the Editor window will always show the next statement to be executed.

The next sections describe some of the debugging windows you can use when you are stopped in the debugger.

Using Edit and Continue

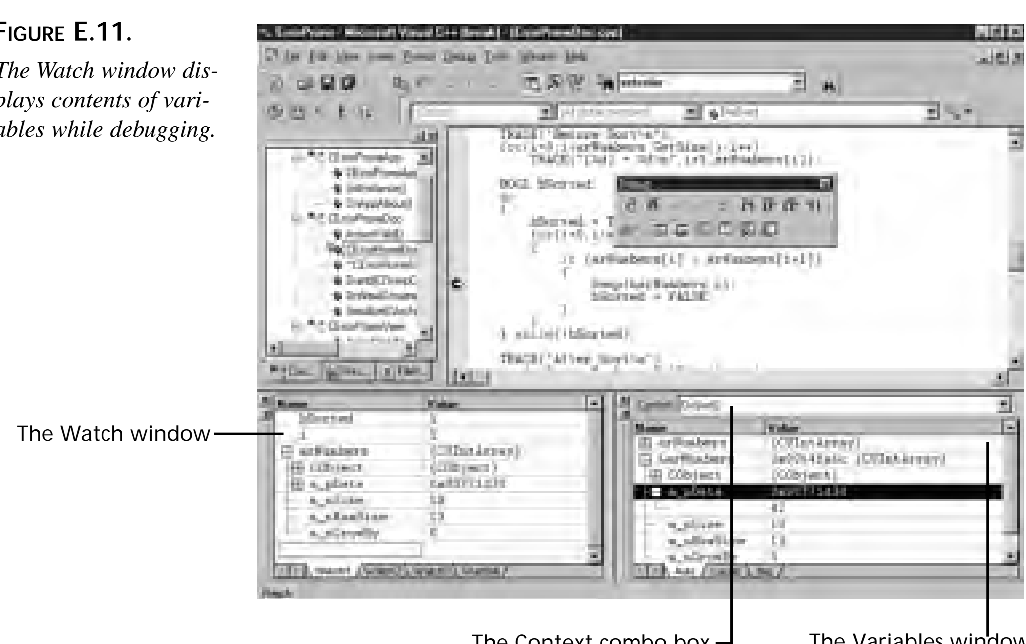
A great new feature of Visual C++ 6 is the capability to Edit and Continue. This means that you can change or edit the code while you are stopped in the debugger. After editing, you'll notice the Debug menu's Apply Code Changes option becomes enabled (as well as the corresponding debug toolbar icon). You can then select the Apply Code Changes option (or toolbar button) to compile your new code changes and then continue debugging the new changed code. By using this new feature, you can fix bugs while debugging and continue the debug run from the same place in the code with the same variable settings, which can be very useful when debugging large and complex programs.

Watching Program Variables

The Watch and Variables windows are shown in Figure E.11. These windows display the contents of variables when stopped in the debugger. You can view these windows by clicking the View menu and selecting them from the Debug Windows pop-up menu or by clicking the icons from the toolbar.

FIGURE E.11.

The Watch window displays contents of variables while debugging.



The Variables window always shows the local variables of the function displayed in the Context combo box at the top of the window. To get to your current function, you can drop this combo box list to display all the functions that were called in turn. This is the call stack and shows your current context within the program by showing the list of functions that have been called in order to get to the program's currently executing function where the debugger has stopped. When you select a different function, the relevant local variables are shown for that function level.

You can expand any object pointers shown by clicking the plus symbol next to the pointer name. The special C++ `this` pointer is always shown for class member functions and can be opened to show all the member variables for the current object.

The Watch window lets you enter variable names from the keyboard or drag variable names from the editor window (after selecting and inverting them with the mouse point). The values that are held in the displayed variables are shown until they go out of scope (that is, aren't relevant to the function currently being debugged).

You can also enter simple casts and array indexes in the Watch window to show related values. Right-clicking the mouse can switch the displayed values between hexadecimal

and decimal display. As you step through the program, the values shown in the Watch and Variable windows are updated accordingly so that you can track how the program changes the variables.

Other Debugger Windows

Other debugging display windows are available by clicking the View menu and selecting them from the Debug Windows pop-up menu or alternatively by clicking the various icons shown to the right of the Debug toolbar. These windows are

- QuickWatch. By clicking a variable in the listing and choosing QuickWatch or pressing Shift+F9, you can display the contents of the select variable. You can also enter variables directly and then click the Add Watch button to transfer them into the main Watch window.
- Registers. The Registers window displays the current values in your CPU's register set. This probably isn't too useful to you unless you are tracking machine or assembly code-level problems.
- Memory. The Memory window displays the memory from the application's address space in columns that represent the address, the hex values, and the character values for each 8 bytes. You can change this display to show Byte, Short, or Long values by right-clicking to display the appropriate context menu options.
- Call Stack. The Call Stack window shows the list of functions that were called in order to get to your current function and the parameter values that were passed to each function. This can be very useful to investigate how the program flow reached a specific function. By double-clicking any of the listed functions, you can display the position where the function call was made in the code, shown by the Editor window.

  Where source code isn't available, function entries are shown as follows:

  ```
  KERNEL32! bff88f75()
  ```

  If you click these entries, you'll be shown assembly code rather than C++ code.
- Disassembly. By selecting the Disassembly toolbar button or menu option, you can toggle between displaying the C++ code mixed with assembly code or just C++ code. Where the source code is unavailable, only assembly code is shown.

Additional Debugging Tools

Along with the integrated debugging tools are several nonintegrated but very useful tools. You can start these by clicking the Tools menu and selecting the specific tool option from the menu.

These tools generally let you track operating-specific items such as Windows messaging, running processes, and registered OLE objects to enhance your available information while debugging your application.

Using Spy++

Spy++ is undoubtedly one of the most useful of these tools. With Spy++, you can see the hierarchical relationships of parent to child windows, the position and flags settings for windows, and base window classes. You can also watch messages as they are sent to a window.

When you first run Spy++, it shows all the windows on the current desktop, their siblings, and the base Windows class of each object (see Figure E.12). The view shown in Figure E.12 has been scrolled to shown the standard Microsoft Windows CD Player. `Spy++` shows you all the buttons and combo boxes, which are windows in their own right as child windows of the main CD Player window.

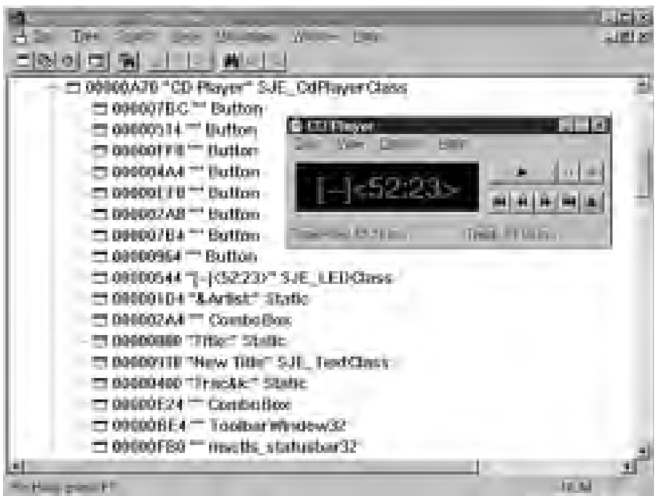


FIGURE E.12.
The Spy++ initial view of the Windows desktop showing the CD Player portion.

If you click the Spy menu, you are shown the following options:

- Messages. You might find that the Messages view is probably one of the most useful options because you can use it to watch messages that are sent to any window (including your own application). You can also filter these messages so that you don't receive an avalanche of Mouse Movement messages.

  To use messages, select this option to display the Message Options dialog box shown in Figure E.13. You can then drag the finder tool over any window in the

system, displaying the details of the window as it moves. Spy++ also highlights the selected window, so you can see frame and client windows. When you've located the window you want to view, just let go of the tool. At this point you can use the other tabs to set filtering options and output formatting options. When you're finished, click OK to close the Message Options box.

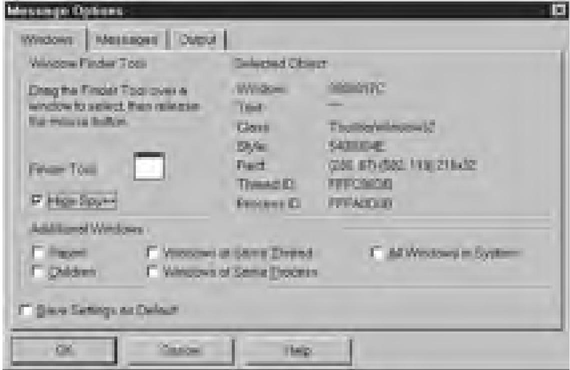


FIGURE E.13.

Using the Spy++ Message Options Finder to locate windows.

The output shown in Figure E.14 are the messages produced from using a normal SDI application's toolbar. As you can see, with no filtering you'll receive many mouse movements and cursor check messages, but you can also see the familiar `WM_LBUTTONUP` message with its position parameters.

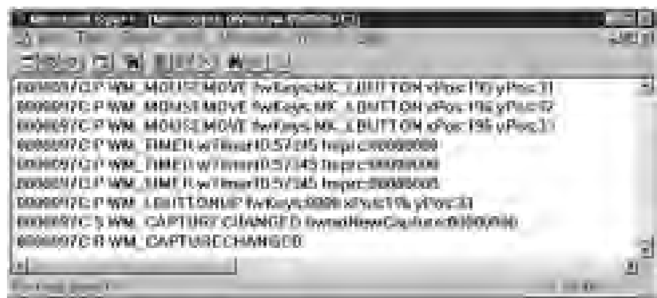


FIGURE E.14.

Windows Messages for a toolbar logged by Spy++.

- Windows. The Windows view is the view shown in Figure E.12 of the layout and structure of the Windows desktop. If you double-click any of these windows, you'll be shown a property sheet containing all the selected windows' positioning information and flag settings. To update this information, you must click the Windows menu and choose Refresh.

- Processes. You can view all the running programs with the Processes view. These can be opened to show each thread and any windows attached to those threads.
- Threads. The Threads option shows the same details without the processes level of hierarchy, so you can see every thread running on your machine and the windows that each thread owns.

Spy++ is too sophisticated to cover in its entirety here, but as a tool for understanding the structure of Windows hierarchies and messaging, it is unsurpassed. You can glean a lot of valuable knowledge just by looking at commercial applications with Spy++. It is also a wonderful tool for debugging messaging problems in your own application to ensure that your windows are getting the correct messages and to see how these messages are sequenced.

Process Viewer

You can see all the processes in more detail than shown in Spy++ with the Process Viewer (`PView95.exe`). You can start this application from your system's main Windows Start menu from Programs under the Microsoft Visual Studio 6.0 Tools option (or similar program group). This application lists the processes running on your machine and lets you sort them by clicking any of the column headers. You can then click a process to display all its threads. Figure E.15 shows Process Viewer running with the Developer Studio application (`MSDEV.EXE`) selected and all its many threads displayed.

FIGURE E.15.

The Process Viewer showing `MSDEV.EXE` *and its threads.*

The OLE/COM Object Viewer

The OLE/COM Object Viewer tool shows you all the registered OLE/COM objects on your system, including ActiveX controls, type libraries, embeddable objects, automation objects, and many other categories.

You can even create instances of various objects and view their interfaces in detail. The OLE/COM Object Viewer is very useful if you are developing an OLE/COM application or looking for an elusive ActiveX control.

The MFC Tracer

Using the MFC Tracer tool shown in Figure E.16, you can stop the normal tracing or add specific Windows trace output to the normal program trace output. When you select this tool, you are shown a set of check boxes that you can check or uncheck to include that tracing option.

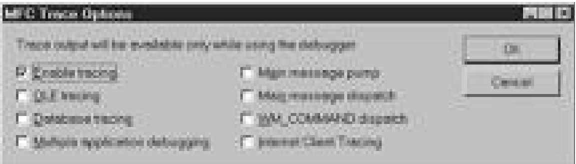


FIGURE E.16.
The MFC Tracer tool options.

You can add Windows messages, database messages, OLE messages, and many other levels of trace output to help track down elusive problems. These messages are then generated by the MFC code for the various selected flags.

You can even turn off the standard tracing generated by your application by unchecking the Enable Tracing option.

APPENDIX F

Using MFC's Helper Classes

by Jon Bates

Using the Collection Classes

You'll quickly find that in any object-oriented program you write, objects must be grouped and stored into collections of different types and sizes. Once again, the MFC comes to the rescue with sets of easy-to-use classes and templates to help with this common requirement.

The collection classes fall into three broad categories—arrays, maps, and lists—that have their particular niches.

Arrays are the mainstay of collection classes and are useful for implementing object containers. Each object in an array has a zero-based position or index, which is used to locate and reference the object.

Lists are useful when you want to think of your data as being linked sequentially, each item to the next. They are useful when you must

quickly add or remove items to the head or tail of the list. You can also traverse the list forward or backward from one item to the next.

Maps are used to associate data against a key item such as a string or number where the associations are sparse and random. For example, you might use a map to associate objects with zip or postal codes. They are good at fast retrieval of objects given the association key and can be used as short-term data caches for large databases.

Using the Array Classes

MFC provides several predefined array classes and a generic array template so that you can create arrays to hold your own customized objects. (The latter is covered later in this chapter, in the "Creating Custom Collection Classes" section.)

Several predefined array classes offer quick and easy array access for common types of variables and objects, as shown in Table F.1.

TABLE F.1. PREDEFINED ARRAY-BASED CLASSES.

Array Class	*Types of Variable Held*	*Numeric Range of Type*
`CByteArray`	`BYTE`—8-bit unsigned values	0 to 255
`CWordArray`	`WORD`—16-bit unsigned values	0 to 65535
`CUIntArray`	`UINT`—32-bit unsigned integer values	0 to 4294967295
`CDWordArray`	`DWORD`—32-bit unsigned integer values	0 to 4294967295
`CStringArray`	`CString`—text string objects	
`CObArray`	`CObject`—any `CObject`-derived objects	
`CPtrArray`	`void*`—any object pointers or memory addresses	

There are several member functions for each array class that differ only by the type of variables they hold. Each function discussed can be used with any array classes to deal with variables of their corresponding type.

One of the most useful aspects of these array classes is their capability to grow dynamically. Normal C/C++ arrays are predefined in size and can be extended only by lots of messy reallocations of memory. The collection classes hide these reallocations so that you can simply call the `Add()` member of an array object to add a new value. For example, to add strings to a `CStringArray`, you can use code similar to this:

```
CStringArray myStringArray;
myStringArray.Add("Red");
myStringArray.Add("Green");
myStringArray.Add("Blue");
```

You can then find the size of an array by calling the `GetSize()` function; for example, the following line used after the previous lines would return three items into `nNumberOfItems`:

```
int nNumberOfItems = myStringArray.GetSize();
```

You can also set the array to a specific size using the corresponding `SetSize()` function, which will extend or truncate the array to the specified size you pass.

Values can be set to the array by using the `SetAt()` function that passes a zero-based index and the value to be stored. `SetAt()` will assert whether the index is larger than the current array size. You can then retrieve values from the array using the `GetAt()` function, which will return the value at the index position that you specify. You might use these functions with a `CWordArray` like this:

```
CWordArray myWordArray;
myWordArray.SetSize(20);
myWordArray.SetAt(0,200);
myWordArray.SetAt(19,500);
TRACE("Value at index position 19 is %d\n",
        myWordArray.GetAt(19));
```

These lines will set the first element of a 20-element array to 200 and the last to 500 and display the value `500` when executed. You can still grow the array by calling the `Add()` function and find the uppermost valid index by calling `GetUpperBound()`, which will return the zero-based index, or –1 if there are no elements present.

You can use the `[ ]` operators to set and get values at a specific index just like a normal C++ array. For example, the `GetAt()` and `SetAt()` functions in the previous lines could be replaced with the `[ ]` operators like this:

```
myWordArray[0] = 200;
myWordArray[19] = 500;
TRACE("Value at index position 19 is %d\n",
        myWordArray.GetAt[19]);
```

Using the `InsertAt()` and `RemoveAt()` functions, you can insert or remove items at a specific position, which results in all the items shifting up or down by one or more elements.

The `InsertAt()` function has two forms; the first needs an index position and an element to insert there. You can also optionally pass it a count to insert multiple copies of the specified element. The second form lets you insert another whole array at a specified index position.

The `RemoveAt()` function needs only one parameter to specify the index value of the item to be removed, but you can also optionally pass a count as the second parameter to remove a number of elements. The remaining array elements will then be shifted down to fill the gap.

You can remove all the elements of an array by calling the `RemoveAll()` function.

> **MANAGING MEMORY WITH `CObArray` AND `CPtrArray`**
>
> You must be careful to delete objects that you have allocated with `new` and stored in a `CObArray` or `CPtrArray` because these arrays only hold pointers to the elements (not elements themselves). Therefore, a `RemoveAll()` call will only remove the pointers to the objects and not free the memory used by the objects themselves.

Using the List Classes

There are only three categories of lists as shown in Table F.2 and a template for your own types (discussed later). There is seldom any need to have a list of simple integer values. Instead, you would probably need a linked list of your own `CObject`-derived classes or pointers to a number of C++ classes or structures.

TABLE F.2. THE LIST-BASED COLLECTION CLASSES.

Class Name	*Type of Variable Held*
`CObList`	`CObject`—Pointers to any `CObject`-derived objects.
`CPtrList`	`void*`—Pointers to memory addresses holding any type of data.
`CStringList`	`CString`—Text strings.

Linked lists are several objects linked to each other in a sequential fashion like carriages on a train. There is a definite head and tail position, but every other element knows only its immediate neighbor. A `POSITION` variable keeps track of a current position in a list. You can declare multiple `POSITION` variables to track different places in the same list.

Each list's member functions then use a `POSITION` variable to find the head, tail, or next and previous elements in the list.

You can add elements to a list by calling the `AddHead()` or `AddTail()` functions or by inserting items into a specific position using the `InsertBefore()` or `InsertAfter()` functions. Each function then returns a `POSITION` value to indicate the position of the new added item.

For example, you can construct a four-element list of `CString` items like this:

```
CStringList listMyStrings;
POSITION pos;
pos = listMyStrings.AddHead("Hand");
listMyStrings.AddTail("Forearm");
listMyStrings.InsertBefore(pos,"Fingers");
listMyStrings.AddTail("Elbow");
```

These lines will produce a linked list of `CString` strings from head to tail like this:

```
Fingers-Hand-Forearm-Elbow
```

You can also pass other similar list objects to the `AddHead()` and `AddTail()` functions to add another list to the head or tail of the current list.

When you've constructed a list, you can iterate through its members using a `POSITION` variable. The head or tail positions can be found by calling `GetHeadPosition()` or `GetTailPosition()`, respectively. These functions both return a `POSITION` value indicating the current position in the list. You can then pass the `POSITION` variable as a reference to `GetNext()` or `GetPrev()` to find the next or previous element in the list. These functions then return the specific object and adjust the current position. When the end of the list is reached, the `POSITION` variable will be set to `NULL`.

For example, the following lines will iterate through the previous `listMyStrings`, displaying each element in turn:

```
POSITION posCurrent = listMyStrings.GetHeadPosition();
while(posCurrent) TRACE("%s\n", listMyStrings.GetNext(posCurrent);
```


You can find specific list elements by using the `Find()` function, which returns a `POSITION` value if the search parameter you pass is found. You can also optionally pass a position value, from which you can start the search.

For example, you can search for the string `Fingers` in the previous list by calling the `Find()` function like this:

```
POSITION posFingers = Find("Fingers");
```

If the searched-for element isn't found, a `NULL` value will be returned.

There is also a `FindIndex()` function that will find the nth element from the head of the list (where `n` is the passed parameter).

You can find out how many elements are in the list by calling the `GetCount()` member function, which returns the number of elements and doesn't need any parameters.

The value of elements at a specific position can be retrieved or reset by using the `GetAt()` and `SetAt()` functions, which are used in a similar way to their array equivalents, but by passing a `POSITION` value rather than an array index.

You can remove elements from the list by using the `RemoveAt()` function and passing the `POSITION` value to identify the element to be removed. For example, to remove the `Fingers` item from the previous example, you might code the following:

```
RemoveAt(posFingers);
```

Using the Map Classes

The map classes work by associating a type value (or element) with a key value that can be used to look up the element. The various map classes, and their key values and associated element types, are shown in Table F.3.

TABLE F.3. THE MAP-BASED COLLECTION CLASSES.

Class Name	*Key Type*	*Element Type*
`CMapWordToOb`	`WORD`—16-bit unsigned value	`CObject`—`CObject`-derived objects
`CMapWordToPtr`	`WORD`—16-bit unsigned value	`void*`—Pointers to memory
`CMapPtrToPtr`	`void*`—Pointers to memory	`void*`—Pointers to memory
`CMapPtrToWord`	`void*`—Pointers to memory	`WORD`—16-bit unsigned value
`CMapStringToOb`	`CString`—Text strings	`CObject`—`CObject`-derived objects
`CMapStringToPtr`	`CString`—Text strings	`void*`—Pointers to memory

Class Name	*Key Type*	*Element Type*
`CMapStringToString`	`CString`—Text strings	`CString`—Text strings

You can insert elements into a map by using the `SetAt()` function and passing a key value as the first parameter and your element value as the second. For example, if you must store your own `CObject`-derived objects indexed by a string value, you can use the `CMapStringToOb` class and add elements like this:

```
CMapStringToOb mapPlanetDetails;
mapPlanetDetails.SetAt("Mercury",new CPlanetDets
  ➥(4878, 0.054, 57.91, 87.969));
mapPlanetDetails.SetAt("Venus",new CPlanetDets
  ➥(12100, 0.815, 108.21, 224.701));
mapPlanetDetails.SetAt("Earth",new CPlanetDets
  ➥(12756, 1.000, 149.60, 365.256));
```

In the previous example, `CPlanetDets` is a `CObject`-derived class with a constructor that takes four planetary detail parameters. The new objects are then associated with the planet names as keys.

You can also use the `[ ]` operator overload instead of `SetAt()` by enclosing the key value inside the square brackets like this:

```
mapPlanetDetails["Mars"] = new CPlanetDets
  ➥(6762, 0.107, 227.94, 686.98);
```

After you have set data to a map, you can retrieve it by calling the `Lookup()` member function by passing the key value and a reference to a variable to hold the associated element value if found. If the element isn't found, a `FALSE` value is returned from `Lookup()`. For example, to retrieve details about a planet from the previous example, you can use these lines:

```
CPlanetDets* pMyPlanet = NULL;
if (mapPlanetDetails.Lookup("Earth",(CObject*&)pMyPlanet))
    TRACE("Sidereal Period = %d days\n", pMyPlanet->m_dSidereal);
```

The `(CObject*&)` cast is used to cast the `pMyPlanet` object pointer to a generic `CObject` pointer reference.

The `GetCount()` function will return the number of elements current in the map. These elements can be removed by calling the `RemoveKey()` function and passing the key of the element to be removed like this:

```
mapPlanetDetails.RemoveKey("Jupiter");
```

Remember to delete the allocated objects. `RemoveKey()` just removes the pointer to the object—not the object itself—so it won't free up the used memory. You can also remove all the elements by calling `RemoveAll()`.

You can iterate through the list of associations using the `GetNextAssoc()` function, which needs parameters that reference a current `POSITION` holding variable, a key variable, and an element variable. You can find the position of the first element by calling `GetFirstPosition()`, which returns the `POSITION` value for the first element. To iterate through the associations, you might code the following:

```
POSITION pos = mapPlanetDetails.GetStartPosition();
while(pos!=NULL)
{
CString strPlanet;
CPlanet* pMyPlanet;
mapPlanetDetails.GetNextAssoc(pos,strPlanet, (CObject*&)pMyPlanet);
TRACE("%s has a diameter of %d km\n",strPlanet, pMyPlanet->m_dDiameter);
}
```

When `GetNextAssoc()` returns, `pos` will hold the position for the next association or `NULL` if there are no more. The key and element values (`strPlanet` and `pMyPlanet` in the previous example) will be set to each key-element pair in turn.

Because of a map's capability to retrieve sparse data quickly and efficiently, it is often advantageous to use a map as a memory cache for a slow database lookup.

For example, in the following lines, the planet details associated with `strPlanetName` are required. When first called, this code won't have a mapped version of the required planet, so it will have to call `GetPlanetFromSlowDB()` to find it. Because it then stores the retrieved planet in the `mapPlanetDetails` map, when it is next called with the same `strPlanetName`, the details can be quickly returned from the cached version in memory:

```
CPlanetDets* pMyPlanet = NULL;
if (mapPlanetDetails.Lookup(strPlanetName,
   ➥(CObject*&)pMyPlanet) == FALSE)
{
pMyPlanet = GetPlanetFromSlowDB(strPlanetName);
mapPlanetDetails.SetAt(strPlanetName,pMyPlanet);
}
return pMyPlanet;
```

This technique is easy to implement and can transform your application's speed when you are using slow retrieval devices such as databases or files.

Creating Custom Collection Classes

You might want to customize the collection classes to use your own objects rather than the generic `CObject`-derived classes. Customization offers several benefits because you can make an array, list, or map that will accept and return only your specific type of object. If you accidentally try to add the wrong sort of object to a customized array, list, or map, the compiler will issue an error message to notify you. The other advantage is that you don't have to cast generic `CObject*` pointers (that is, from a `CObArray`) back to your specific object to use it.

This sort of customization is known as *type-safety*; in large programs it can be invaluable for stopping accidental assignments of the wrong class. A set of templates, `CArray`, `Clist`, and `CMap`, lets you easily create an array, list, or map to store, use, and return objects of your specified type only. Templates are a complex subject, but you don't have to worry about writing templates; the MFC-provided templates defined in the `afxtempl.h` header file will do for these type-safe collection classes. For the scope of this section, it is best to think of templates as large macros that generate lots of code based on your parameters when compiled.

The templates will give you access to all the normal functions in the array, list, or map classes discussed in the previous sections. However, instead of using generic `CObject`-based parameters and returned values, you can define your own types as parameters and return values.

To use the templates in your program, you'll need to include the following header line in each module (`.cpp`/`.h` file) that uses the template definitions:

```
#include "afxtempl.h"
```

You can then define your own custom type-safe class using the template syntax like this for an array of custom objects:

```
CArray<CMyCustomClass*, CMyCustomClass *> myCustomClassArray;
```


The `<` and `>` symbols used in the definition should be thought of as angle brackets (not greater-than or less-than conditional operators). The previous line uses the `CArray` template to create an instance of `myCustomClassArray`. The first `CMyCustomClass*` parameter specifies types of object pointers you want the array to return when you use `GetAt()` and other access functions. The second `CMyCustomClass*` specifies the type that should be used for the input parameter definitions. Then all the functions that store objects, such as `SetAt()` and `Add()`, will accept only pointers to objects of your specific `CMyCustomClass`.

For example, you can create an array that takes and returns only pointers to the specific `CPlanetDets` class, defined (and implemented) like this:

```
class CPlanetDets : public CObject
{
public:
CPlanetDets(double dDiameter,double dGravity,
➥double dDistFromSun,double dSidereal):
    m_dDiameter(dDiameter), m_dGravity(dGravity),
    m_dDistFromSun(dDistFromSun), m_dSidereal(dSidereal) {}
  double m_dDiameter,m_dGravity,m_dDistFromSun,m_dSidereal;
};
```

To declare a type-safe `CArray`-based array called `myPlanetArray`, you can then code the following line:

```
CArray<CPlanetDets*,CPlanetDets*> myPlanetArray;
```

This declares that `myPlanetArray` can only accept pointers to a `CPlanetDets` object and return pointers to a `CPlanetDets` object. You might then use the new array like this:

```
myPlanetArray.Add(new CPlanetDets
  ➥(4878, 0.054, 57.91, 87.969));
myPlanetArray.Add(new CPlanetDets
  ➥(12100, 0.815, 108.21, 224.701));
myPlanetArray.Add(new CPlanetDets
  ➥(12756, 1.000, 149.60, 365.256));
for(int i=0;i<myPlanetArray.GetSize();i++)
   TRACE("Diameter = %f\n", myPlanetArray[i]->m_dDiameter);
```

These lines create three new `CPlanetDets` type objects and add them to the array. The last line displays the diameter in the `TRACE` macro without needing to cast the returned value from `myPlanetArray[i]` because it's already a pointer of the `CPlanetDets*` type.

However, later you might forget the exact nature of `myPlanetArray` and try to add a `CStatic` object instead:

```
myPlanetArray.Add(new CStatic());
```

Fortunately, the compiler will spot the transgression and issue a compiler error such as

```
'Add' : cannot convert parameter 1 from 'class
➥CStatic *' to 'class CPlanetDets *
```

However, the error wouldn't have been spotted if you had been using a `CObArray` to hold the planet details:

```
CObArray myPlanetArray;
```

The `CStatic` object would be happily stored along with the `CPlanetDets` objects, causing untold havoc when you try to retrieve the `CStatic` object, thinking it's a `CPlanetDets` object.

The template used to generate type-safe lists is `CList`; it takes the same general form as `CArray`:

```
CList<CMyCustomClass*, CMyCustomClass *> myCustomClassList;
```

Again, the first parameter is the required returned object type, and the second parameter specifies the accepted object types for functions that accept elements for storage.

All the functions available for lists are available for your own specific type-safe customized lists, again checking and returning your specified types. Therefore, the equivalent list-based code for the planet storing array would be coded like this:

```
CList<CPlanetDets*,CPlanetDets*> myPlanetList;
myPlanetList.AddTail(new CPlanetDets
  ➥(4878, 0.054, 57.91, 87.969));
myPlanetList.AddTail(new CPlanetDets
  ➥(12100, 0.815, 108.21, 224.701));
myPlanetList.AddTail(new CPlanetDets
  ➥(12756, 1.000, 149.60, 365.256));
POSITION pos = myPlanetList.GetHeadPosition();
while(pos) TRACE("Diameter = %f\n",myPlanetList.
  ➥GetNext(pos)->m_dDiameter);
```

The template for customized maps differs from the list and arrays in that it needs four parameters: an input and a return value for both the key and element value. So the general form is like this:

```
CMap<MyType, MyArgType, CMyCustomClass *, CMyCustomClassArg *>
myCustomClassMap;
```

The first parameter, `MyType`, specifies the internally stored key value for each map association. This can be any of the basic types such as `int`, `WORD`, `DWORD`, `double`, `float`, or `CString`, or it can be a pointer to your own specific type.

The second parameter, `MyArgType`, specifies the argument type used to pass key values in and out of the map functions.

The third parameter, `CMyCustomClass *`, is how you want the internal element values to be stored (as specific type-safe pointers to your objects).

The fourth parameter, `CMyCustomClassArg *`, specifies the argument type used to pass your element values in and out of the map functions. For example, to associate the planet details with their names, you might code the following:

```
CMap<CString,LPCSTR,CPlanetDets*,CPlanetDets*> myPlanetMap;
myPlanetMap.SetAt("Mercury",
          new CPlanetDets(4878, 0.054, 57.91, 87.969));
myPlanetMap.SetAt("Venus",
          new CPlanetDets(12100, 0.815, 108.21, 224.701));
myPlanetMap.SetAt("Earth",
          new CPlanetDets(12756, 1.000, 149.60, 365.256));
CPlanetDets* pPlanet = NULL;
if (myPlanetMap.Lookup("Venus",pPlanet))
TRACE("Diameter = %f\n",pPlanet->m_dDiameter);
```

The map declaration indicates that the objects should be stored internally as `CStrings` but use `LPCSTR` (pointers to constant character arrays) to pass values into and out of the map. The planet's details themselves will be both stored internally and accessed as pointers to `CPlanetDets` objects (such as `CPlanetDets*`).

POTENTIAL PROBLEMS WHEN USING THE MAP'S INTERNAL HASH KEY TYPES

You must be wary of the conversion between the passed parameters and the internal hash key storage system. For example, if you were to replace the `CString` in the previous example with another `LPCSTR` for the internal storage object, the `Lookup()` would fail to find `"Venus"` because it would be comparing the pointer values (to different instances of `"Venus"`) rather than the contents of the strings.

Using the Coordinate-Handling Classes

Because Windows is a graphically oriented environment, you'll often need to hold point positions, rectangles, and sizes. Three MFC classes help store and manipulate these coordinates: `CPoint`, `CRect`, and `CSize`. Each has several member functions and operator overloads that take much of the work out of adding, constructing, and finding derivatives of these coordinates.

Also several of the MFC and GDI functions understand their types or underlying types as parameter values, so you don't have to perform any messy mangling operations to pass them into functions.

Using the `CPoint` Class

`CPoint` encapsulates a `POINT` structure that just holds an `x` and `y` position to represent a point on a two-dimensional surface. You can always access `x` and `y` members directly to get or set their current values like this:

```
CPoint ptOne;
ptOne.x = 5;
ptOne.y = 20;
TRACE("Co-ordinate = (%d,%d)\n",ptOne.x,ptOne.y);
```

You set these values when you construct a `CPoint` object by passing values to one of `CPoint`'s several constructors, as shown in Table F.4.

TABLE F.4. CONSTRUCTOR TYPES FOR THE `CPoint` CLASS.

Constructor Definition	*Description*
`CPoint()`	Constructs an uninitialized object
`CPoint(POINT ptInit)`	Copies the settings from a `POINT` structure or another `CPoint` object
`CPoint(int x, int y)`	Initializes the object from the `x` and `y` parameter values
`CPoint(DWORD dwInit)`	Uses the low 16 bits for the `x` value and the high 16 bits for the `y` value
`CPoint(SIZE sizeInit)`	Copies the settings from a `SIZE` structure or `CSize` object

For example, you could replace the last sample lines with these for the same result:

```
CPoint ptOne(5,20);
TRACE("Co-ordinate = (%d,%d)\n",ptOne.x,ptOne.y);
```

One of the most useful aspects of the `CPoint` class is its many operator overloads. By using the `+`, `-`, `+=`, and `-=` operators with other `CPoint`, `CRect`, or `CSize` objects, you can add or subtract coordinate pairs from other coordinate pairs or from rectangles or sizes. For example, the long way to subtract two points from each other to give a third would be like this:

```
CPoint ptOne(5,20);
CPoint ptTwo(25,40);
CPoint ptThree;
ptThree.x = ptTwo.x – ptOne.x;
ptThree.y = ptTwo.y – ptOne.y;
```

This can be simplified by using the operator overload:

```
CPoint ptOne(5,20);
CPoint ptTwo(25,40);
CPoint ptThree = ptTwo – ptOne;
```

Or you can add the coordinates of one point to another like this:

```
ptTwo += ptOne;
```

You can also use the == and != logical operator overloads to perform comparisons. For example, to check whether ptTwo is equal to ptOne in both x and y values, you can code the following:

```
if (ptOne == ptTwo) TRACE("Points are the same");
```

There is also an Offset() function that adds an offset value specified by passing x and y values, or a CPoint class or POINT structure, or a CSize or SIZE structure. Therefore, the following two lines are functionally identical:

```
ptOne.Offset(75,-15);
ptOne-=CPoint(-75,15);
```

Using the CRect Class

The CRect class encapsulates a RECT structure to hold two pairs of coordinates that describe a rectangle by its top-left point and its bottom-right point. You can construct a CRect object using one of its several constructors, as shown in Table F.5.

TABLE F.5. CONSTRUCTOR TYPES FOR THE CRect CLASS.

Constructor Definition	*Description*
CRect()	Constructs an uninitialized object
CRect(const RECT& rcInit)	Copies the settings from another RECT structure or CRect object
CRect(LPCRECT lprcInit)	Copies the settings via a RECT or CRect pointer
CRect(int l,int t,int r,int b)	Initializes the coordinates from left, top, right, and bottom parameters
CRect(POINT point, SIZE size)	Initializes from a POINT or CPoint and a SIZE or CSize
CRect(POINT ptTL, POINT ptBR)	Initializes from a top-left POINT and a bottom-right POINT

After you've constructed a CRect object, you can access each of the top, left, bottom, and right members individually using the (LPRECT) cast to cast it into a RECT structure as shown in these lines:

```
CRect rcOne(15,15,25,20);
 ((LPRECT)rcOne)->bottom += 20;
TRACE("Rect is (%d,%d)-(%d,%d)",
((LPRECT)rcOne)->left,((LPRECT)rcOne)->top,
        ((LPRECT)rcOne)->right,((LPRECT)rcOne)->bottom);
```

Alternatively, you can access the members via either the top-left CPoint or the bottom-right CPoint. References to these member objects are returned by the TopLeft() and BottomRight() functions. When you've accessed either the top-left or bottom-right

points, you can then manipulate them using any of the `CPoint` functions shown in the previous section. For example, the following lines are functionally identical to the previous lines, but differ in that they construct and access the rectangle using `CPoint` objects:

```
CRect rcOne(CPoint(15,15),CPoint(25,20));
rcOne.BottomRight().y += 20;
TRACE("Rect is (%d,%d)-(%d,%d)",
    rcOne.TopLeft().x,rcOne.TopLeft().y,
    rcOne.BottomRight().x,rcOne.BottomRight().y);
```

You can also use the `SetRect()` function to set the coordinates by passing four integers to represent the top-left `x`- and `y`-coordinates and the bottom-right `x`- and `y`-coordinates. `SetRectEmpty()` sets all these coordinates to zero to make a `NULL` rectangle. The `IsRectNull()` function will return `TRUE` if called on such a `NULL` rectangle, and `IsRectEmpty()` will return `TRUE` if the width and height are both zero (even if the individual values are not zero).

Several helper functions help you calculate various aspects of the rectangle's geometry. The width and height can be found by calling the `Width()`and `Height()` functions, each of which returns the relevant integer value. Alternatively, you can find a `CSize` that represents both width and height by calling the `Size()` function. For example, the following line displays the width and height of the rectangle `rcOne`:

```
TRACE("Rect Width = %d, Height = %d\n",
                    rcOne.Width(), rcOne.Height());
```

The point in the center of the rectangle is often a useful coordinate to know; you can find this by calling the `CenterPoint()` function, which returns a `CPoint` object to represent the center of the rectangle.

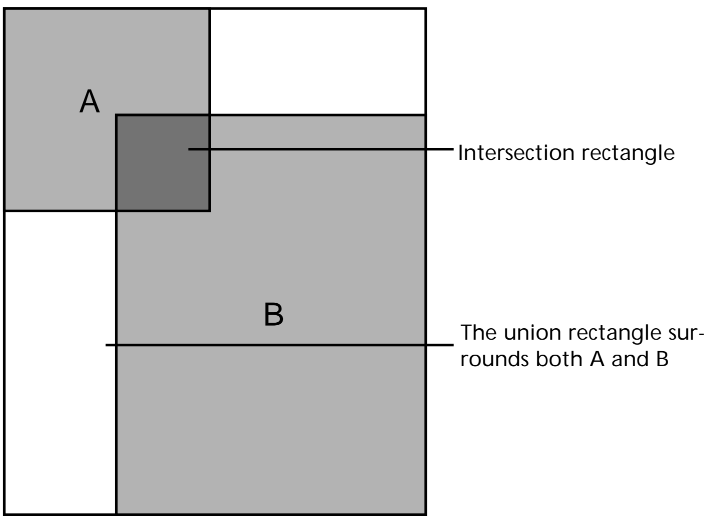
You might use this to find the center of your window's client area and draw a dot there like this:

```
CRect rcClient;
GetClientRect(&rcClient);
dc.SetPixel(rcClient.CenterPoint(),0);
```

You can also find the union or intersection of two rectangles by calling `UnionRect()` and `InterSectRect()`, which both take two source rectangles as parameters and set the coordinates of the calling `CRect` object to the union or intersection. The union is the smallest rectangle that will enclose the two source rectangles. The intersection is the largest rectangle that is enclosed by both source rectangles. The diagram in Figure F.1 shows the union and intersection of two source rectangles labeled `A` and `B`.

FIGURE F.1.

The union and intersection between two rectangles.



The following lines calculate the intersection and union of the source rectangles `rcOne` and `rcTwo`:

```
CRect rcOne(10,10,100,100);
CRect rcTwo(50,50,150,200);
CRect rcUnion, rcIntersect;
rcUnion.UnionRect(rcOne,rcTwo);
rcIntersect.IntersectRect(rcOne,rcTwo);
```

When this code is run, `rcUnion` will be set to coordinates (10,10)–(150,200) and `rcIntersect` will be set to coordinates (50,50)–(100,100).

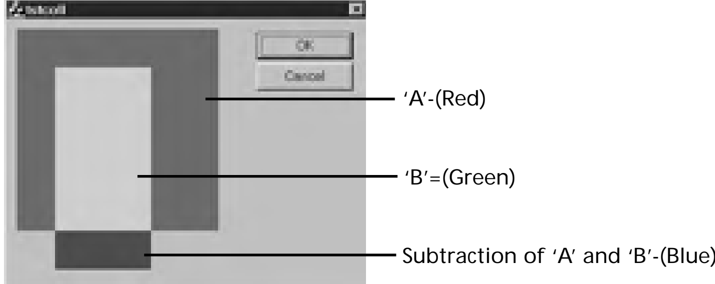
You can use `SubtractRect()` to find the subtraction of one rectangle from another. This is the smallest rectangle that contains all the points not intersected by the two source rectangles (or the smallest non-overlapping section). For example, by adding the following lines to an `OnPaint()` handler, you can see the effects of `SubtractRect()` to subtract `rcTwo` from `rcOne` to produce `rcDst`. The resulting subtraction is the section that will be drawn in blue at the bottom of the diagram, as shown in Figure F.2.

```
CRect rcOne(10,10,220,220), rcTwo(50,50,150,260), rcDst;
rcDst.SubtractRect(rcTwo,rcOne);
dc.FillSolidRect(rcOne,RGB(255,0,0));//red
dc.FillSolidRect(rcTwo,RGB(0,255,0));//green
dc.FillSolidRect(rcDst,RGB(0,0,255));//blue
```

When this code is run, the resulting rectangle `rcDst` will hold the coordinates (50,220)–(150,26).

FIGURE F.2.

The effects of a subtraction operation on two partially overlapping rectangles.



You can increase or decrease the size of a rectangle using `InflateRect()` and `DeflateRect()`. These both have several forms that accept various types of parameters, as shown in Table F.6.

TABLE F.6. PARAMETER FORMS FOR `InflateRect` AND `DeflateRect`.

Parameters	*Description*
`(int x, int y)`	Inflate or deflate the left and right sides by the `x` value and the top and bottom sides by the `y` value.
`(SIZE size)`	Inflate or deflate the left and right sides by `size.cx` and the top and bottom sides by `size.cy`.
`(LPCRECT lpRect)`	Inflate each side by the corresponding `left`, `top`, `right`, and `bottom` values from `lpRect`.
`(int l, int t, int r, int b)`	Inflate each side by the corresponding left, top, right, and bottom values.

For example, the following code inflates `rcOne` and deflates `rcTwo`:

```
CRect rcOne(10,10,100,100);
CRect rcTwo(50,50,150,200);
rcOne.InflateRect(5,5);
rcTwo.DeflateRect(10,20,30,40);
```

After these lines have run, `rcOne` will be set to the coordinates (5,5)–(105,105) and `rcTwo` will be set to the coordinates (60,70)–(120,160).

You can perform hit-testing by determining whether a specified point (perhaps from a mouse click) lies within the bounds of a rectangle by calling `PtInRect()` and passing the point to be tested. If the point does lie within the rectangle, a `TRUE` value is returned; otherwise a `FALSE` value is returned.

In the following lines, the `Hit! - ptTest1` message is displayed because `ptTest1` does lie within the `rcTestArea` test area, whereas `ptTest2` doesn't, so `PtInRect()` returns `TRUE` for `ptTest1` and `FALSE` for `ptTest2`:

```
CRect rcTestArea(10,20,440,450);
CPoint ptTest1(200,200), ptTest2(500,500);
if (rcTestArea .PtInRect(ptTest1))
  ➥AfxMessageBox("Hit! - ptTest1");
if (rcTestArea .PtInRect(ptTest2))
  ➥AfxMessageBox("Hit! - ptTest2");
```

There are also several operator overloads for use with `CRect` objects, as shown in Table F.7.

TABLE F.7. OPERATOR OVERLOADS USED WITH `CRect`.

Operator	*Description*
`=`	Copies all the coordinates from the right rectangle operand to the left rectangle, like an ordinary numeric assignment.
`+`	Either displaces a rectangle position if a `CPoint` or `CSize` object is added to a rectangle or inflates the coordinates with their corresponding counterparts if a `CRect` object is added.
`-`	Same as `+`, except that the coordinates are displaced in a negative direction or deflated if a `CRect` is used.
`+=`	Same overall effect as `+` but affects only the current rectangle.
`-=`	Same overall effect as `-` but affects only the current rectangle.
`&`	Creates an intersection rectangle from the two rectangle operands.
`¦`	Creates a union rectangle from the two rectangle operands.
`&=`	Same overall effect as `&` but affects only the current rectangle.
`¦=`	Same overall effect as `¦` but affects only the current rectangle.
`==`	Returns `TRUE` if the rectangles are identical, otherwise `FALSE`.
`!=`	Returns `FALSE` if the rectangles are identical; otherwise returns `TRUE`.

The following lines show some of the `CRect` operator overloads being used to manipulate the `rcStart` rectangle:

```
CRect rcStart(10,10,100,100);
rcStart = rcStart + CPoint(5,5);
rcStart -= CSize(5,5);
rcStart    += CRect(1,2,3,4);
if (rcStart == CRect(9,8,103,104)) AfxMessageBox("TRUE");
```

The final condition returns TRUE, thus displaying the message box because the final coordinates are (9,8)–(103,104).

> **USING THE NormalizeRect() FUNCTION**
> Sometimes you might perform an operation that makes the top-left point hold values greater than the bottom-right point. If this is so, the width or height might be negative, causing other functions to fail. If you suspect this might happen, you can call the NormalizeRect() function to correct the coordinates so that the top-left coordinates have lower values than the bottom-right coordinates.

Using the CSize Class

The CSize class encapsulates the SIZE structure and provides several constructors and operator overloads that manipulate the internal cx and cy values that define a size. The various constructors you can use to create an instance of a CSize object are shown in Table F.8.

TABLE F.8. CONSTRUCTOR TYPES FOR THE CSize CLASS.

Constructor Definition	*Description*
CSize()	Creates an uninitialized CSize object.
CSize(SIZE sizeInit)	Copies the cx and cy values from another CSize object or SIZE structure.
CSize(initCX, initCY)	Initializes the object with initCX for the horizontal size and initCY for the vertical size.
CSize(POINT ptInit)	Initializes the object with the x and y values from a CPoint object or POINT structure.
CSize(DWORD dwSize)	Sets the cx value to the low-word (bottom 16 bits) of dwSize and the cy value to the high-word (to 16 bits) of dwSize.

You can manipulate the cx and cy members directly like this:

```
CSize tstSize(10,10);
tstSize.cx = tstSize.cy * 2;
```

The only functions that the CSize class offers are operator overloads, as shown in Table F.9.

F

TABLE F.9. OPERATOR OVERLOADS USED WITH `CSize`.

Operator	*Description*
+	Add two size objects
-	Subtract one size object from another
+=	Add a `SIZE` object
-=	Subtract a `SIZE` object
==	Determine whether the two sizes are the same and return `TRUE` if identical
!=	Determine whether the two sizes are different and return `TRUE` if different

These can be used just like normal arithmetic operators and affect both the `cx` and `cy` members, as shown in the following lines that manipulate the contents of `tstSize`:

```
CSize tstSize(10,15);
tstSize += tstSize + tstSize - CSize(1,2);
if (tstSize == CSize(29,43)) AfxMessageBox("TRUE");
```

When run, this code will display the `TRUE` message box message because `tstSize` ends up as the size 29×43.

Using the Time-Handling Classes

The capability to store dates and times is a common requirement for many applications. You will probably also need to calculate elapsed times and time spans between stored date and time values and be able to format those into user-readable text strings.

MFC provides four classes to handle all the aspects of date and time manipulation and storage. Originally, there were just two classes; `CTime` and `CTimeSpan`, which are based on the UNIX `time_t` (4 byte long value) system (the number of elapsed seconds since 1970). However, granularity of only one second and a limited range of dates between 1970 and 2038 proved too restrictive for many applications. Hence, two new replacement classes, `COleDateTime` and `COleDateTimeSpan`, are now also supplied and should be used in preference to `CTime` and `CTimeSpan` in newer applications.

`COleDateTime` is based on an underlying `DATE` structure (which is actually just a `double` value). This greater capacity of storage type lets `COleDateTime` cover a range of dates between January 1, 100, and December 31, 9999, and down to an approximate resolution of 1 millisecond. The difference between two `COleDateTime` values can be represented and manipulated by the `COleDateTimeSpan` object.

Because of the similarity between the `CTime` class and the newer `COleDateTime` class, the following sections just describe `COleDateTime`, although many of the functions are identical in the `CTime` versions.

USING CTime WITH DATABASES

You might find it convenient to use CTime when using ODBC-based databases because the RFX recordset transfer macros know only how to handle CTime objects directly and don't know how to handle COleDateTime objects without conversion. If you use DAO databases, COleDateTime can be used directly.

Using the COleDateTime Class

COleDateTime is connected with OLE in that it can be used in conjunction with the VARIANT structure, often used in OLE automation. Because of the wide range of date and time storage systems, especially in OLE environments, COleDateTime must be capable of converting between all these various types. This support is reflected in its many constructor forms, as shown in Table F.10.

TABLE F.10. CONSTRUCTOR TYPES USED WITH COleDateTime.

Constructor Definition	*Description*
COleDateTime()	Creates an uninitialized COleDateTime object
COleDateTime(const COleDateTime& datesrc)	Copies the values from another COleDateTime object
COleDateTime(int nYear, int nMonth, int nDay, int nHour, int nMinute, int nSecond)	Initializes the date and time from the values passed
COleDateTime(const VARIANT& varSrc)	Converts a date time from a VARIANT structure
COleDateTime(DATE dtSrc)	Copies a date time from a DATE structure
COleDateTime(time_t timeSrc)	Copies a date time from a UNIX-style time_t structure
COleDateTime(WORD wDosDate)	Copies a date time from the MS-DOS–style values WORD and wDosTime
COleDateTime(const SYSTEMTIME& systimeSrc)	Copies a date time from a SYSTEMTIME structure
COleDateTime(const FILETIME& filetimeSrc)	Copies a date time from a FILETIME structure

If you've constructed `COleDateTime` with a valid date time, the object will be marked with a valid status flag (`COleDateTime::valid`). Otherwise, the status flag is invalid (`COleDateTime::invalid`). You can check this status by calling the `GetStatus()` member function to return the relevant flag value, or you can force the flag value by passing it into the `SetStatus()` function.

The status flag is also updated when you set date and time values to the object by calling the `SetDateTime()` function. This function takes six integer parameters for the year (100–9999), month (1–12), day of the month (1–31), hour (0–23), minute (0–59), and second (0–59). You can also set just the date or time components by calling `SetDate()`—passing just the year, month, and day of the month—or by calling `SetTime()`—passing only the hour, minute, and second values.

You can use the `GetCurrentTime()` static function to retrieve the current system time and set it to a `COleDateTime` object using the = operator overload function like this:

```
COleDateTime dtCurrent;
dtCurrent = COleDateTime::GetCurrentTime();
```

After running these lines, `dtCurrent` will be set to your machine's current system date and time.

The same values (in the same ranges) can be retrieved by the return value of `GetYear()`, `GetMonth()`, `GetDay()`, `GetHour()`, `GetMinute()`, or `GetSecond()`. There are also the useful derivative functions: `GetDayOfWeek()` and `GetDayOfYear()`. `GetDayOfWeek()` returns the day of the week in the range 1 to 7 where 1 is Sunday. The `GetDayOfYear()` function returns a value in the range 1 to 366 starting at January 1.

You can retrieve a displayable formatted `CString` by using the `Format()` function. This is probably one of the most useful `COleDateTime` functions because you can pass several formatting codes to specify the exact format returned from `Format()`, as shown in Table F.11. These codes are passed as either a string or a string resource identifier, and several individual codes are strung together to add various aspects of the formatting.

These values are also modified by the current locale settings. The locale preferences can affect things such as the names of days and months, the ordering of MM/DD/YY representations, and the AM/PM indicators.

TABLE F.11. FORMATTING CODES TO FORMAT THE `COleDateTime` TEXT OUTPUT.

Code	*Example*	*Description*
`%a`	Sat	Abbreviated day of the week
`%A`	Saturday	Day of the week
`%b`	Apr	Abbreviated month

Code	*Example*	*Description*
`%B`	April	Month
`%c`	04/04/98 18:05:01	Date and time in the current locale format
`%d`	04	Day of the month (01–31)
`%H`	18	Hour (00–23) 24 hour
`%I`	06	Hour (01–12) 12 hour
`%j`	094	Day of the year
`%m`	04	Month (01–12)
`%M`	05	Minute (01–59)
`%p`	PM	AM/PM indicator for locale
`%S`	01	Second (01–59)
`%U`	13	Week of year (00–51) with Sunday as first day of week
`%w`	6	Weekday (0–6) 0=Sunday
`%W`	13	Week of year (00–51) with Monday as first day of week
`%x`	04/04/98	Date in the current locale format
`%X`	18:05:01	Time in the current locale format
`%y`	98	Two-digit year (00–99)
`%Y`	1998	Four-digit year (0100–9999)
`%z` or `%Z`	GMT Daylight Time	Time zone name/abbreviation
`% %`	%	A percent sign
`%#c`	Saturday, April 04, 1998 18:05:01	Current locale long date and time
`%#x`	Saturday, April 04, 1998	Current locale long date

F

You can use the following lines to generate a message box displaying your machine's date and time like this:

```
COleDateTime dtCurrent;
dtCurrent = COleDateTime::GetCurrentTime();
AfxMessageBox(dtCurrent.Format("Today is %a %b %d, %Y"));
```

When run, the time will then be displayed in a message box in the following format:

```
Today is Sat Apr 04, 1998
```

You can get COleDateTime to attempt to determine a date and time by calling ParseDateTime() and passing a string for it to parse and a flag value to specify that only the date or the time component is required. ParseDateTime() will then scan the string for time in the format HH:MM:SS, and a date in the format DD/MM/YYYY, or in a long format such as January 18th, 1998. If you only want to scan for the time, you can pass VAR_TIMEVALUEONLY for the second parameter flag value, alternatively VAR_DATEVALUEONLY for just the date. If you don't want the users' locale preferences to be used to indicate the string format to check for, you can pass LOCALE_NOUSEROVERRIDE as this flag value.

There are also several operator overloads you can use to add and subtract COleDateTimeSpans and to compare date and time values with other date and time values as shown in Table F.12.

TABLE F.12. OPERATOR OVERLOADS USED IN COleDateTime.

Operator	*Description*
=	Copy a date/time value from another COleDateTime object, VARIANT structure, DATE structure, time_t structure, SYSTEMTIME structure, or FILETIME structure.
+	Add a COleDateTimeSpan value to a COleDateTime value.
-	Subtract a COleDateTimeSpan from a COleDateTime value or two COleDateTime objects from each other to yield a COleDateTimeSpan result.
+=	Add a COleDateTimeSpan value to the current COleDateTime object.
-=	Subtract a COleDateTimeSpan value from the current COleDateTime object.
==	Check whether two COleDateTime objects hold an identical date and time.
!=	Check whether two COleDateTime objects hold different dates and times.
<	Check whether one COleDateTime object is less than another.
>	Check whether one COleDateTime object is greater than another.
<=	Check whether one COleDateTime object is less than or equal to another.
>=	Check whether one COleDateTime object is greater than or equal to another.

Using the COleDateTimeSpan Class

A COleDateTimeSpan object can hold the difference between two COleDateTime objects. You can create one by subtracting one COleDateTime object from another or by using one of the COleDateTimeSpan constructor forms shown in Table F.13.

TABLE F.13. CONSTRUCTOR TYPES USED WITH `COleDateTimeSpan`.

Constructor Definition	*Description*
`COleDateTimeSpan()`	Create a time span set to zero.
`COleDateTimeSpan(const COleDateTimeSpan& srcSpan)`	Copy the time span from another `COleDateTimeSpan` object.
`COleDateTimeSpan(long lDays, int nHours, int nMins, int nSecs)`	Initialize the time span with the passed parameter values.
`COleDateTimeSpan(double dSpanSrc)`	Initialize the time span with the number of days passed value.

After you have a `COleDateTimeSpan` object, you can check or set its status using the `GetStatus()` and `SetStatus()` functions just like the `COleDateTime` object. The only differences are that the flag values are `COleDateTimeSpan::valid` and `COleDateTimeSpan::invalid`.

You can also set a time span by passing the number of days, hours, minutes, and seconds as integer parameters to `SetDateTimeSpan()`. You can then retrieve these values from a valid `COleDateTimeSpan` object by calling the `GetDays()`, `GetHours()`, `GetMinutes()`, and `GetSeconds()` functions that all return long values representing each portion of the time span value. If you want to retrieve the overall time span expressed in days, hours, minutes, or seconds in one double value, you can call `GetTotalDays()`, `GetTotalHours()`, `GetTotalMinutes()`, or `GetTotalSeconds()`, respectively.

You can format `COleDateTimeSpan` values as strings in the same way as `COleDateTime` values by passing a format string using the codes appropriate to time spans from Table F.11.

Several operator overloads help you use `COleDateTimeSpan` objects arithmetically to add and subtract time spans to and from each other and also use them in conditions, as shown in Table F.14.

TABLE F.14. OPERATOR OVERLOADS USED IN `COleDateTimeSpan`.

Operator	*Description*
=	Copies time spans from other time span values
+	Adds two time spans
-	Subtracts one time span from another

continues

TABLE F.14. CONTINUED

Operator	*Description*
`+=`	Adds a time span to the current object
`-=`	Subtracts a time span from the current object
`==`	Checks to see whether two time spans are identical
`!=`	Checks to see whether two time spans are different
`<`	Checks to see whether one time span is less than another
`>`	Checks to see whether one time span is greater than another
`<=`	Checks to see whether one time span is less than or equal to another
`>=`	Checks to see whether one time span is greater than or equal to another

The following sample code shows how two `COleDateTime` objects can be subtracted to yield a `COleDateTimeSpan` using the minus (`-`) operator overload:

```
COleDateTime dtMoonwalk;
dtMoonwalk = COleDateTime(1969,7,20,0,0,0);
COleDateTimeSpan dtDiff =
              COleDateTime::GetCurrentTime()- dtMoonwalk;
CString strMessage;
strMessage.Format("Days since first moonwalk: %d ",
                             (int)dtDiff.GetTotalDays());
AfxMessageBox(strMessage);
```

Using the String Manipulation Class

For several years C programmers secretly envied one (and only one) tool that BASIC programmers had at their disposal: sophisticated and easy string handling. With C++, that functionality can be naturally replicated, and has been with MFC's `CString` class.

String handling is a very common application requirement, and Visual C++ applications tend to be littered with instances of `CString`-based objects to accomplish the task.

Using the `CString` Class

You can easily construct `CString` objects as an empty string or initialized by passing one of many differing types of text representation systems to the constructor. The various forms of `CString` construction are shown in Table F.15.

TABLE F.15. CONSTRUCTOR TYPES USED WITH `CString`.

Constructor Definition	*Description*
`CString()`	Creates an empty zero-length string
`CString(const CString& strSrc)`	Copies the contents from another `CString`
`CString(LPCSTR lpsz)`	Copies the contents from a null-terminated string
`CString(const unsigned char* psz)`	Copies the contents from a null-terminated string
`CString(LPCTSTR lpch, int nLength)`	Copies `nLength` characters from a character array
`CString(TCHAR ch, int nRepeat = 1)`	Fills the string with `nRepeat` copies of character `ch`
`CString(LPCWSTR lpsz)`	Copies a Unicode null-terminated string

When you've constructed a `CString` object, there are many ways in which you can add or assign text to it. The operator overloads provide simple assignments via the = operator, or concatenation of two strings by the + or += operators, as shown in the following lines:

```
CString strTest;
strTest = "Mr Gorsky";
strTest = "Good luck " + strTest;
AfxMessageBox(strTest);
```

When this code is run, the string is initially set to `"Mr Gorsky"`; then `"Good luck "` is prefixed by the + operator.

You can find the length of a string by calling the `GetLength()` member function, which returns an integer representing the number of characters the string currently holds. You can also test whether a string is empty by calling `IsEmpty()`, which returns `TRUE` if it holds no characters.

You can also drop the contents of a `CString` object by calling `Empty()`, which will then force the string to be zero length.

Many functions require old C-style strings rather than a `CString` object; you can use the `(const char *)` or `LPCTSTR` casts to give functions access to the `CString` object's internal buffer as if it were a C-style null-terminated string (but only for read access). Visual C++ will implicitly cast the `CString` to a null-terminated string if a specific function has a prototype that requires it. However, because some functions might have a `void*` prototype, the compiler will pass a pointer to the `CString` object rather than the null-terminated string it expects, thus requiring you to perform the `(LPCTSTR)` cast on the `CString` object.

Using NULL-Terminated Strings

NULL-terminated strings are character arrays that use a `NULL` or zero value character to mark the end of the string. Therefore the length of the string is the number of characters before the `NULL` character. There are several C-style functions such as `strlen()`, `strcpy()` (find the length and copy a string), and many others that are used to manipulate NULL-terminated strings.

You can access the string as an array of characters by using the `GetAt()` and `SetAt()` functions. `GetAt()` will return the character at a specified (zero-based) position parameter. `SetAt()` will set a character (second parameter) to a position (first parameter) within the length of the string. You can also use the `[ ]` operator to get a character value from a specific position instead of `GetAt()`. For example, the following lines transpose the `e` and `i` characters to fix the spelling mistake:

```
CString strText("Fix this spilleng");
TCHAR ch1 = strText.GetAt(11);
strText.SetAt(11,strText[14]);
strText.SetAt(14,ch1);
```

You can also use the conditional operator overloads `<`, `<=`, `==`, `!=`, `>=`, and `>` to test one string lexicographically against another. The ASCII codes are compared when the strings are compared, so numbers are less than letters and uppercase letters are less than lowercase letters. Therefore, the following lines of code will cause the `TRUE` message box to appear:

```
CString str1("123");
CString str2("ABC");
CString str3("abc");
CString str4("bcd");
if (str1 < str2 && str2 < str3 && str3 < str4)
                                     AfxMessageBox("TRUE");
```

You can also use the `Compare()` function to compare the current string with another. This will return zero if the two strings are equal, a negative value if the current string is less than the tested string, or a positive value if the current string is greater than the tested string. For example, the following lines will cause the `TRUE` message box to appear:

```
CString strName("Peter");
if (strName.Compare("Piper")<0) AfxMessageBox("TRUE");
```

This comparison also takes notice of the case of the compared strings. You can use the `CompareNoCase()` equivalent to ignore differences in case.

String Manipulation

The BASIC language had three very useful manipulation functions—`Mid$`, `Left$`, and `Right$`—which are now available as the `CString` `Mid()`, `Left()`, and `Right()` functions. Their job is to copy out sections of a specified string. You can pass the `Mid()` function a start position and optionally a number of characters to copy (it defaults to all characters if not specified), and `Mid()` will return another `CString` containing the specified section. You can extract a number of characters from the left of a string by passing the required count to the `Left()` function. `Right()` returns the corresponding specified number of characters from the right of the string as shown in these lines:

```
CString strText("I saw three ships a sailing");
TRACE("%s\n",strText.Left(5));
TRACE("%s\n",strText.Mid(6,11));
TRACE("%s\n",strText.Right(7));
```

When run, these lines will display the following trace output:

```
I saw
three ships
sailing
```

Notice that the word *a* used between ships and sailing isn't displayed in the trace output because that portion of the string is never extracted.

You can use the `MakeUpper()` and `MakeLower()` functions to change all the characters in a string to upper- or lowercase and `MakeReverse()` to reverse the string.

Spaces, newline characters, and tabs can be trimmed from the left of a string with the `TrimLeft()` function and from the right with the `TrimRight()` function.

Searching Strings

You can look for specific substrings or characters from a string by using the `Find()`, `ReverseFind()`, or `FindOneOf()` functions.

You can pass a single character or string to the `Find()` member function to look for an instance of that character or string in the context string. If the character or substring is found, its zero-based position will be returned; otherwise –1 is returned to indicate that the substring or character isn't present. The following lines search for the word `"you"` and display the substring starting at that position:

```
CString strTest("Start as you mean to go on");
int nPosn = strTest.Find("you");
if (nPosn!=-1) TRACE(strTest.Mid(nPosn) + "?");
```

`ReverseFind()` searches for a specified character (no substring form) from the end of a string and if found, returns its position from the beginning of the string; otherwise it returns –1.

`FindOneOf()` lets you pass a number of characters to look for in a string. The position of the first character from the set to be found is returned, or –1 is returned to indicate that none were found. The following sample lines search for the characters `g`, `m`, `i`, and `p`; the `m` is found first, so the string `"mean to go on"` is displayed in the trace output:

```
CString strTest("Start as you mean to go on");
int nPosn = strTest.FindOneOf("gmip");
if (nPosn!=-1) TRACE(strTest.Mid(nPosn));
```

Formatting Text for Display

A common use for `CString` objects is for formatting text before it is displayed. You can use the `Format()` function for this formatting by passing a set of formatting instructions as `%` coded instruction strings as the first parameter. Then you must pass a number of value parameters that correspond to each of those instructions. Some of the possible formatting instruction flags and corresponding parameter types are shown in Table F.16. Several of these codes can be combined to form a formatted string, which is then stored in the calling `CString` object. This format string can also be passed as a string resource ID from your application's resources.

TABLE F.16. FORMATTING `%` INSTRUCTION CODES FOR USE WITH THE `Format()` FUNCTION.

Flag	*Parameter Type*	*Description*
`%c`	`int`	Displays a single text character
`%d`	`int`	Displays a signed decimal integer
`%u`	`int`	Displays an unsigned decimal integer
`%o`	`int`	Displays an unsigned octal integer
`%x`	`int`	Displays an unsigned hexadecimal integer in lowercase
`%X`	`int`	Displays an unsigned hexadecimal integer in uppercase
`%f`	`double`	Displays a signed floating-point number
`%s`	`string`	Displays a string of text when passed a string pointer such as `char*`
`%%`	`none`	Display a percent sign %.

The following example combines some of these common types to display a message box containing the text `"There are 1.609340 Kilometers to 1 Mile"`:

```
CString strFormatMe;
char *szKm = "Kilometer";
double dKmsPerMile = 1.60934;
strFormatMe.Format("There are %f %ss to %d %s",
                    dKmsPerMile,szKm,1,"Mile");
AfxMessageBox(strFormatMe);
```

You can pass extra formatting specification flags to the width and precision of each output field and its alignment. These flags are passed after the `%` sign, but before the type (refer to Table F.16) in the following format:

```
%[flags] [width] [.precision]type
```

The flag value can be any of the following characters:

- `-` Left-aligns the field (otherwise it is right-aligned).
- `+` Displays a + or – to indicate the numeric sign (defaults to just show – for negative numbers).
- **`0`** Pads the width with zeroes

The `width` then specifies the minimum number of characters that should be displayed, and the `.precision` specifies how many characters should be displayed after a floating point.

For example, you could change the text from the previous example to read `"There are +1.6 Kilometers to 1 Mile"` by changing the format for `%f` to `%+1.f` like this:

```
strFormatMe.Format("There are %+.1f %ss to %d %s",
                    dKmsPerMile,szKm,1,"Mile");
```

STRING FORMATTING

The `CString Format()` function uses the same format specifier codes as the C-style `printf()` and `sprintf()` (print and string print) functions. C programmers might be more familiar with the formatting codes of these `printf()` functions, which can also be used and applied to the `CString Format()` function.

Index

Symbols

& (ampersand), 31
- address-of operator, 554-555
- binary AND (&), 51-53
- logical AND (&&), 51

= (equal sign), 43, 549

*** (asterisk)**
- indirection operator, 554
- multiplication operator, 549

{ } (braces), 546

// (double slashes), 545

>> (extraction operator), 548

| (pipe character)
- binary OR (|), 52-53
- logical OR (||), 52

<< (redirection operator), 546

:: (scope operator), 568

. (structure member operator), 561

~ (tilde), 564

A

AbortDoc function, 657

About window, editing, 23

ABOVE_NORMAL thread priority, 432

accelerators
- creating, 118-119
- defined, 107
- restrictions, 119
- Web Browser application, 532

Accept function, 501

accessing
- field values, 353-354
- structures, 561

ActiveX controls
- adding to dialogs, 186-187
- automation, 181
- COM, 180-181
- containers, 182-183
- defined, 180, 474
- event handling, 193-194, 476
- FlexGrid
  - cell IDs, 189
  - loading data into, 188-189
  - random data generation, 189-191
  - sorting, 192-193
- Internet Explorer support, 522
- licensing issues, 184, 196, 477
- methods, 475-476
- properties
  - adding, 475
  - ambient, 474
  - custom, 474
  - extended, 474
- registering
  - Registered ActiveX Controls folder, 184-186
  - regsvr32 utility, 183
- Squiggle
  - CModArt class, 478-481
  - control shell, 477-478

event handling, 486-487
events, 489-491
methods, 487-489
properties, 481-483
property page, 483-485
testing, 491-492
see also ADO
ActiveX Data Objects, *see* ADO
Add Member Function dialog box, 18
Add Member Variable dialog box, 32
Add Property dialog box, 475
Add function, 701
AddHead function, 703
Adding a Class dialog box, 113
AddLine function, 206
AddNew function, 320, 357, 374
AddNewRecord function, 296
address-of operator (&), 554-555
AddString function, 264
AddTail function, 703
ADO (ActiveX Data Objects), 344-345
clean-up functions, 367
command execution, 351
Command object, 345-346
Connection object, 345
database connections
closing, 357
creating, 350-352, 365-366
DLLs, importing, 349-350
error handling, 364
Error object, 345
Field object, 346
forms, populating, 367-369
macros
ADO_FIXED_LENGTH_ENTRY, 355
ADO_NUMERIC_ENTRY, 355
ADO_VARIABLE_LENGTH_ENTRY, 355-356
BEGIN_ADO_BINDING, 354
END_ADO_BINDING, 354
Parameter object, 346
pointers, 364
records
adding, 357, 373-374
creating, 374-375
custom record classes, 361-364
deleting, 357, 375-376
saving, 370-371
updating, 356
recordsets
accessing, 353-354
closing, 357
navigating, 352-353, 371-372
RecordSet object, 346
see also DbAdo application
ADO_FIXED_LENGTH_ENTRY macro, 355
ADO_NUMERIC_ENTRY macro, 355
ADO_VARIABLE_LENGTH_ENTRY macro, 355-356
AfxBeginThread function, 432-433, 468
AfxGetStaticModuleState function, 409
AFX_EXT_CLASS macro, 408
AFX_MANAGE_STATE macro, 409
aligning objects, 450-452
combo boxes, 263
toolbars, 251
ALTDRAG toolbar control style, 250
ambient properties, 474
ampersand (&), 31
address-of operator, 554-555
binary AND (&), 51-53
logical AND (&&), 51
AND operators
binary AND (&), 51-53
logical AND (&&), 51
ANSIOTROPIC mapping mode, 156
appendOnly flag (recordsets), 318
applications
controls
adding, 27-30
functions, 35
initialization code, 33-35
tab order, 30-31
variables, 32-33
flow, tracking, 685-687
menus
accelerators, 107, 118-119
attaching, 112
cascading, 106
context menus, 106, 116-118
creating, 109-110
event handling, 112-115
hotkeys, 107
naming, 119
pull-down menus, 106
separator lines, 111
Windows conventions, 107
Multiple Document Interface, *see* MDI applications
project workspace, creating, 11-12
resource files, 108
running, 543-544

Single Document Interface, *see* SDI applications
toolbars, 244-245
attaching to application frames, 247-249
borders, 251
buttons, 245, 252-253
child window IDs, 251
creating, 246, 250
docking, 253-254
hiding, 254-256
loading, 250
naming, 247
styles, 250-251
views, 8
see also names of individual applications; listings
AppWizard
application frameworks, creating, 12-14
printing/previewing, 637
arguments, passing by reference, 557-558
arithmetic operators, 549
arrays
classes, 700-702
Spinner application, 449-450
threads, 433
aspect ratio
controlling, 645-647
fax machines, 645
printers, 645
ASSERT macro, 687-689
assignment operator (=), 549
attaching
menus, 112
toolbars to application frames, 247-249
attribute flags (binary operations), 52-53
AUTOHSCROLL combo box style, 263
automation, 181

B

Back button (browser application), 531
BeginWaitCursor function, 63
BEGIN_ADO_BINDING macro, 354
BELOW_NORMAL thread priority, 432
binary operators
AND (&), 51-52
attribute flags, 52-53
OR (|), 52
bItalic argument (CreateFont function), 127
BitBlt function, 155
bitmaps
copying, 155
displaying, 174-176
loading, 155
LoadBitmap function, 172
LoadImage function, 172
OnBbitmap function, 173-174
resizing, 155
variables, 173
Black menu (SDI drawing application), 224
Blue menu (SDI drawing application), 224-225
borders (toolbars), sizing, 251
BottomRight function, 712
braces ({}), 546
breakpoints, 685
setting, 689-690
toggling on/off, 690
Breakpoints dialog box, 690
browsers
Internet Explorer
ActiveX document model, 522
controlling, 524
navigation functions, 523-524
status, 524
WebBrowse application
application shell, 525
Back button, 531
displaying URLs, 529-530
Forward button, 531
Refresh button, 533-535
specifying URLs, 526-528
Stop button, 532-533
browsing code (Source Browser)
base classes, 682
Call Graph option, 683
Callers Graph option, 683
derived classes, 683
File Outline option, 682
for definitions/references, 681
identifiers, 681
brushes
creating, 153
patterns, 154
selecting, 153
Build toolbar, 10
bUnderline argument (CreateFont function), 127
button combination IDs (MessageBox function), 85
buttons
adding to toolbars, 245
Exit, creating, 36-37
icons, 246
maximize/minimize, 21-22
radio buttons, 27

removing from toolbars, 245
Run Program, 42-43
styles, 252-253
text, 276
WebBrowse application
Back, 530-532
Forward, 530-532
Refresh, 533-535
Stop, 532-533

C

cache, 704-706
CalcPoint function, 451
calculating
cell IDs, 189
object positions, 450-452
call stack
size, 433
viewing, 693-694
Call Stack window (debugger), 694
callback functions, 72, 122
canceling print jobs, 657
canvases (graphics), creating, 160
captions, 110
capturing
keyboard events, 57-62
mouse events
OnLButtonDown function, 56
OnMouseMove function, 49-50, 54-55
CArchive class, 280
CArchiveException class, 671-672
cascading menus, 106
CAsyncSocket class, *see* sockets
catch keyword, 663-664
catching exceptions, 661-666
blanket exception catch, 665
catch keyword, 663-664
multiple exceptions, 664-665
try keyword, 663-664
CBitmap class, 154-156
CBRS_ALIGN_ANY toolbar style, 251-253
CBRS_ALIGN_BOTTOM toolbar style, 251-253
CBRS_ALIGN_LEFT toolbar style, 251-253
CBRS_ALIGN_RIGHT toolbar style, 251-253
CBRS_ALIGN_TOP toolbar style, 251
CBRS_BORDER_BOTTOM toolbar style, 251
CBRS_BORDER_LEFT toolbar style, 251
CBRS_BORDER_RIGHT toolbar style, 251
CBRS_BORDER_TOP toolbar style, 251
CBRS_FLOAT_MULTI toolbar style, 251-253
CBRS_FLYBY toolbar style, 251
CBRS_GRIPPER toolbar style, 251
CBRS_TOOLTIPS toolbar style, 251
CBrush class, 153-154
CBS_AUTOHSCROLL combo box style, 263
CBS_DISABLENOSCROLL combo box style, 264
CBS_DROPDOWN combo box style, 263
CBS_DROPDOWNLIST combo box style, 264
CBS_HASSTRINGS combo box style, 264
CBS_NOINTEGRALHEIGHT combo box style, 264
CBS_OEMCONVERT combo box style, 264
CBS_OWNERDRAWFIXED combo box style, 264
CBS_OWNERDRAWVARIABLE combo box style, 264
CBS_SIMPLE combo box style, 264
CBS_SORT combo box style, 264
CByteArray class, 700
CChildFrame class, 231
CCriticalSection class, 436
CDAOException class, 672-673
CDbAdoDoc class
AddNew function, 374
CreateBlankRecord function, 374-375
Delete function, 376
DeleteContents function, 367
GenerateError function, 365
GetRecSet function, 364
OnNewDocument function, 366
CDbAdoView class
MoveFirst function, 372
OnDataFirst function, 371
RefreshBoundData function, 368-369
UpdateBoundData function, 370-371
CDBException class, 672-673
CDbOdbcSet constructor, 331
CDocument class, 200
CDWordArray class, 700
CEditView class, 201
cell IDs (FlexGrid control), calculating, 189
CenterPoint function, 713

CEvent class, 440
CException class
Delete function, 668
extending, 675-676
GetErrorMessage function, 668-669
ReportError function, 669
CFileDialog class
properties, 102
sample application
DoModal function, 91
GetFileName function, 92
OnFileopen function, 90-91
CFileException class, 667, 670-671
CFormView class, 201
CFrameView class, 200
changing, *see* editing
char data type, 548
check boxes, 27
CHtmlView class, 201
browser controls, 524
browser status, 524
navigation functions, 523-524
circles, drawing, 168-171
Class View, 8
Class Wizard
Member Variables tab, 32
Message Maps tab, 34
classes
CArchive, 280
CArchiveException, 671-672
CBitmap, 154-156
CBrush, 153-154
CByteArray, 700
CChildFrame, 231
CCriticalSection, 436
CDAOException, 672-673
CDbAdoDoc
AddNew function, 374
CreateBlankRecord function, 374-375
Delete function, 376
DeleteContents function, 367
GenerateError function, 365
GetRecSet function, 364
OnNewDocument function, 366
CDbAdoView
MoveFirst function, 372
OnDataFirst function, 371
RefreshBoundData function, 368-369
UpdateBoundData function, 370-371
CDBException, 672-673
CDocument, 200
CDWordArray, 700
CEditView, 201
CEvent, 440
CException
Delete function, 668
extending, 675-676
GetErrorMessage function, 668-669
ReportError function, 669
CFileDialog, 90-92
CFileException, 667, 670-671
CFormView, 201
CFrameView, 200
CHtmlView, 201
browser controls, 524
browser status, 524
navigation functions, 523-524
CLine
color, 218-219
constructor, 204-205
creating, 203
drawing, 205
header file, 216
serialization, 217
CListView, 201
CMainFrame, 231
CreateColorbar function, 261-262
message map, 268
OnCreate function, 248-249, 265-266
OnSelChangeWidth function, 267
OnUpdateViewColorbar function, 255
OnViewColorbar function, 256
UpdateWidthCB function, 269
CMapPtrToPtr, 704
CMapPtrToWord, 704
CMapStringToOb, 704
CMapStringToPtr, 704
CMapStringToString, 705
CMapWordToOb, 704
CMapWordToPtr, 704
CMemoryException, 669
CModArt, *see* CModArt class
CMutex, 436
CMyCustomClass*, 707
CNotSupportedException, 674
CObArray, 205, 700
CObList, 702
COleDispatchException, 673-674
COleException, 673-674
collection classes, 699-700
array classes, 700-702
customizing, 707-710

list classes, 702-704
map classes, 704-706
type-safety, 707-710
constructors, 564
CPen, 151-153
CPerson, 288
constructor, 290
inline functions, 289-290
serializing, 291-294
variables, 289
CPoint class, 712
CPrintDialog, 653
CPtrArray, 700
CPtrList, 702
CRecordset
database connections, 317
editing functions, 320-321
recordset navigation, 319-320
CRect class, 712-717
CResourceException, 670
CRichEditView, 201
CScrollView, 201
CSemaphore, 438
CSize class, 717-718
CSpinner
color table, 445-446
constructor, 444
declaration, 445
Draw function, 446-447
source file, 449
variables, 443-444
CStringArray, 700
CStringList, 702
CTaskingApp, 457-459
CTaskingDoc
CalcPoint function, 451
DoSpin function, 455
InitSpinners function, 453
OnNewDocument function, 454-455, 469
source file, 450
SuspendSpinner function, 462-463, 470-471
ThreadFunc function, 461, 470
CTaskingView
OnCbthread1 function, 464
OnCbthread2 function, 465
OnDestroy function, 467
CTestAppDoc
declaration, 423-424
DeleteContents function, 425
GetDrawing function, 425
OnNewDocument function, 424
Serialize function, 425
CTestAppView, 425-426
CToolbarDoc
message map, 274
UpdateColorbar function, 269-270
CTreeView, 201
CUIntArray, 700
CUserException, 674
CView, 200
CWinApp, 200
CWordArray, 700
declaring, 385, 561-564
destructors, 564
device context, 151
encapsulation, 380
friend classes, 565
generic, 381-382
header files, 565-566
inheritance, 380-381
Car class example, 572-576
Human class example, 569-572
nesting, 566-569
objects, adding, 708-709
serializable classes, 283-284
string manipulation class, 724
adding text, 725
case sensitivity, 727
constructor types, 724
formatting, 728-729
manipulation functions, 727
NULL-terminated types, 726
searches, 727-728
size, 725
time-handling classes, 718-719
COleDateTime class, 719-722
COleDateTimeSpan class, 722-724
formatting code, 720-721
ClearDrawing function, 391
clearing
drawings
ClearDrawing function, 391
ModArtClearDrawing function, 421
message box text, 38-39
CLine class
color, 218-219
constructor, 204-205
creating, 203
drawing, 205
header file, 216
serialization, 217
CListView class, 201
clock timer
OnInitDialog function, 71-72
OnTimer function, 73
Close function, 319, 357, 503

closing
applications, 36-37
database connections, 357
recordsets
ADO, 357
ODBC, 319
socket connections, 503, 517-519
CMainFrame class, 231
functions
CreateColorbar, 261-262
OnCreate, 248-249, 265-266
OnSelChangeWidth, 267
OnUpdateViewColorbar, 255
OnViewColorbar, 256
UpdateWidthCB, 269
message map, 268
CMapPtrToPtr class, 704
CMapPtrToWord class, 704
CMapStringToOb class, 704
CMapStringToPtr class, 704
CMapStringToString class, 705
CMapWordToOb class, 704
CMapWordToPtr class, 704
CMemoryException class, 669
CModArt class, 478-480
library module
ClearDrawing function, 391
color table, 392
constructor, 391
creating, 385
Draw function, 390
#include directives, 392
NewDrawing function, 389, 397-398
NewLine function, 386, 399
rand function, 388
Serialize function, 390
SetRect function, 386
variables, 385
MFC DLL
class declaration, 411
NewLine function, 411-416
testing, 413-414
regular DLL, 416
header file, 417
ModArtClearDrawing function, 421-422
ModArtDraw function, 421
ModArtNewDrawing function, 418
ModArtSerialize function, 421
module definition file, 422
NewLine function, 419-420
project file, 417
CMutex class, 436
CMyCustomClass* class, 707
CNotSupportedException class, 674
CObArray class, 205, 700
CObList class, 702
code listings, *see* listings
CoInitialize function, 350
COleDateTime class, 722
COleDispatchException class, 673-674
COleException class, 673-674
Collate check box (Print dialog box), 651
collection classes, 699-700
array classes, 700-702
customizing, 707-710
list classes, 702-704
map classes, 704-706
type-safety, 707-710
color
brushes, 153
CModArt class color table, 392
menus
event handlers, 235
troubleshooting, 240
pens, 152-153
SDI application
color table specification, 219-220
lines, 218-222
GetColor function, 222
specifying, 51
Spinner application, 445-446
COM (Component Object Model), 180, 684
see also ActiveX controls
combo boxes
creating
Create function, 263
CreateColorBar function, 260-262
OnCreate function, 265-266
styles, 263-264
event handling
CMainFrame message map, 267-268
OnSelChangeWidth function, 267
placeholders, 262
populating, 264
positioning, 263
resource files, editing, 257-260
sizing, 276
updating, 268-270
command buttons, *see* buttons
COMMAND function, 227
Command object, 345-346

commands
- Control menu, Invoke Methods, 492
- Edit menu, Insert New Control, 491
- executing, 351
- Layout menu, Tab Order, 30
- regsvr32, 184
- *see also* statements

CommandToIndex function, 252

comments, 545

common dialogs
- adding to applications, 90-92
- classes, 90
- OnFileopen function, 91
- *see also* dialog windows

Compare function, 726

CompareNoCase function, 726

compiler
- configuration modes, 678
- #import directive, 349-350
- #include directive, 97
  - CModArt, 392
  - CModArt class, 392
  - CTestAppDoc, 394, 414
  - DbAdo application, 364
  - Graphics application, 162
  - Hello, World application, 545
  - SDI application, 213
  - Sock application, 510
  - Squiggle control, 481

Component Object Model, *see* COM

components, *see* controls

conditional statements, 549

configuring
- combo box placeholders, 262
- compiler modes, 678
- ODBC data sources, 322-323

Connect function, 500

Connection object, 345

constructors
- declaring, 564
- default, 564
- defined, 564
- naming, 564
- return values, 564

containers, 182-183, 700-702

context menus
- creating, 116, 236-238
  - Menu Designer, 237-238
  - OnContextMenu function, 117, 238-239
  - TrackPopupMenu function, 117-118
  - WM_CONTEXTMENU message, 237
- defined, 106
- designing, 116
- event handling, 116

CONTEXTMENU event, 116

Control menu commands, Invoke Methods, 492

control styles (toolbars), 250

controls (ActiveX)
- adding to dialogs, 186-187
- automation, 181
- COM, 180-181
- containers, 182-183
- defined, 180, 474
- event handling, 193-194, 476
- FlexGrid
  - cell IDs, 189
  - loading data into, 188-189
  - random data generation, 189-191
  - sorting, 192-193
- Internet Explorer support, 522
- licensing issues, 184, 196, 477
- methods, 475-476
- properties
  - adding, 475
  - ambient, 474
  - custom, 474
  - extended, 474
- registering
  - Registered ActiveX Controls folder, 184-186
  - regsvr32 utility, 183
- Squiggle
  - CModArt class, 478-481
  - control shell, 477-478
  - event handling, 486-487
  - events, 489-491
  - methods, 487-489
  - properties, 481-483
  - property page, 483-485
  - testing, 491-492

controls (Visual Basic components), 25-26
- adding to applications, 27-30
- buttons, 27
  - Exit, 36-37
  - Run Program, 42-43
- check boxes, 27
- drop-down list boxes, 27
- edit boxes, 26
  - clearing text, 38-39
  - disabling, 39-41
  - displaying text, 37-38
  - hiding, 39-41
- functions, 35
- initialization code, 33-35
- mnemonics, 31
- object IDs, 44-45
- radio buttons, 27
- static text, 26
- tab order, specifying, 30-31
- variables, assigning, 32-33

converting DLLs for non-C++ applications, 427-428

cooperative multitasking, 430

coordinate handling classes, 710
CPoint class, 712
CRect class, 712-717
CSize class, 717-718
Offset function, 712
operator overloads, 711-712
coordinate systems
aspect ratio, 645
mapping modes, 156
copying bitmaps, 155
CoUninitialize function, 350
counting lines, 207-208
counting timer
control variables, 74-75
functions
OnChangeInterval, 75
OnInitDialog, 76
OnTimer, 77-79
starting, 76-77
Stop button, 79-81
stopping, 77
CPen class, 151-153
CPerson class, 288
constructor, 290
inline functions, 289-290
serializing
DECLARE_SERIAL macro, 292
IMPLEMENT_SERIAL macro, 293-294
Serialize function, 291-292
variables, 289
.cpp file extension, 565
CPrintDialog class, 653
CPrintInfo function, 642
CPtrArray class, 700
CPtrList class, 702
CreateBlankRecord function, 374-375
CreateColorbar function, 261-262
CreateEx function, 250-251
CreateFont function
alternatives to, 139
arguments
bItalic, 127
bUnderline, 127
cStrikeOut, 127
lpszFacename, 129
nCharSet, 127-128
nClipPrecision, 128
nEscapement, 126
nHeight, 126
nOrientation, 126
nOutPrecision, 128
nPitchAndFamily, 129
nQuality, 128
nWeight, 126-127
nWidth, 126
declaring, 125
example, 125
CreateHatchBrush function, 655
CreateInstance function, 350
CreateProcess function, 44
CRecordset class
database connections, 317
recordsets
closing, 319
editing, 320-321
navigating, 319-320
opening, 317-318
CResourceException class, 670
CRichEditView class, 201
CScrollView class, 201
CSemaphore class, 438
CSpinner class
color table, 445-446
constructor, 444
declaration, 445
Draw function, 446-447
source file, 449
variables, 443-444
cStrikeOut argument (CreateFont function), 127
CStringArray class, 700
CStringList class, 702
CTaskingApp class, 457-459
CTaskingDoc class
functions
CalcPoint, 451
DoSpin, 455
InitSpinners, 453
OnNewDocument, 454-455, 469
SuspendSpinner, 462-463, 470-471
ThreadFunc, 461, 470
source file, 450
CTaskingView class
OnCbthread1 function, 464
OnCbthread2 function, 465
OnDestroy function, 467
CTestAppDoc class
declaration, 423-424
document functions
DeleteContents, 425
GetDrawing, 425
OnNewDocument, 424
Serialize, 425
CTestAppView class, 425-426
CToolbarDoc class
message map, 274
UpdateColorbar function, 269-270
CTreeView class, 201
CUIntArray class, 700
cursors, changing, 59
CUserException class, 674
custom properties, 474
CUSTOMERASE toolbar control style, 250
customizing
collection classes, 707-710
dialog windows, 93-96
header files, 96-98
variables, 99-100
Print dialog box, 652
toolbars, 10, 245

CView class, 200-201
CWinApp class, 200
CWordArray class, 700

D

DAO (Data Access Objects), 672-673
data sources (ODBC), configuring, 322-323
data types
- declaring, 559
- sizes, 554-555
- values, 547-548

databases, 284
- cache, 704-706
- clean-up functions, 302, 367
- command execution, 351
- connections
  - DbAdo example, 365-366
  - closing, 357
  - ODBC, 317
  - opening, 350-351
- data source configuration, 322-323
- error handling, 364
- exception classes
  - CDAOException, 672-673
  - CDBException, 672-673
- opening, 351-352
- position counters, 297-298
- records
  - adding, 295-297, 331-334, 357, 373-374
  - class headers, 310
  - counting, 297
  - creating, 374-375
  - custom record class, 361-363
  - deleting, 334-335, 357, 375-376
  - displaying, 306-307
  - saving updates, 308-312, 370-371
  - updating, 356
- recordsets
  - closing, 319, 357
  - editing, 320-321
  - navigating, 298-301, 307-308, 319-320, 352-353, 371-372
  - opening, 317-318
  - pointers, 364
  - types, 318
- views, 285
  - resetting, 309-310
  - view class, 305-306
- *see also* DbAdo application; DbOdbc application

date/time handling classes, 718-719
- COleDateTime class, 719-722
- COleDateTimeSpan class, 722-724
- formatting code, 720-721

DbAdo application (ADO database)
- #include directives, 364
- application shell, 358
  - control properties, 359-361
  - control variables, 361
- clean-up functions, 367
- connecting to, 365-366
- form, populating, 367-369
- navigation
  - MoveFirst function, 372
  - OnDataFirst function, 371
- pointers, 364
- records
  - adding, 373-374
  - creating, 374-375
  - custom record class, 361-363
  - deleting, 375-376
  - error handling, 364
- saving updates, 370-371

DbOdbc application (ODBC database)
- application shell, 324-325
- main form
  - CDbOdbcSet constructor, 331
  - control properties, 325-327
  - control variables, 328
  - Database field variables, 329
  - DoDataExchange function, 330
- records
  - adding, 331-334
  - deleting, 334-335

Debug Assertion Failed dialog box, 687
Debug Info, 679
Debug mode, 678
debugger
- call stack, viewing, 693-694
- Disassembly toolbar, 694
- Edit and Continue options, 692
- windows
  - Memory, 694
  - QuickWatch, 694
  - Registers, 694
  - Variables, 692-694
  - Watch, 692-694

debugging techniques, 677
- ASSERT macro, 687-689
- breakpoints, 685
- Debug mode, 678
- just-in-time, 684
- MFC Tracer tool, 698
- networks, 685
- OLE/COM Object Viewer tool, 697
- options/levels, 679-681

Debugging Information settings, 680-681
Generate Browse Info, 679
Preprocessor Definitions, 680
Project Options, 680
warning levels, 679
Warnings as Errors, 679
Process Viewer tool, 697
remote debugging, 683-684
single stepping, 685
breakpoints, 689-690
editing code, 692
step options, 691-692
Source Browser tool
base classes option, 682
Call Graph option, 683
Callers Graph option, 683
Definitions and References option, 681
derived classes option, 683
File Outline option, 682
identifiers, entering, 681
Spy++ tool
Messages view, 695-696
Processes view, 697
Threads view, 697
Windows view, 696
TRACE macros, 685-687
variables, checking contents, 692-694
VERIFY macro, 687-689
see also exceptions; troubleshooting

DECLARE_SERIAL macro, 282, 292

declaring
classes, 385, 561-564
data types, 559
functions, 546-548
constructors, 564
destructors, 564
friends, 565
prototypes, 548
pointers, 554
variables, 548

DEF files, 408

default constructors, 564

defined flags, 52

DeflateRect function, 715

Delete function, 320-321, 376, 668

DeleteContents function, 214, 302, 367, 396, 425

deleting
drawings, 396
DeleteContents function, 425
SDI application, 214
exceptions, 668
records
ADO, 357, 375-376
ODBC, 321, 334-335

derivation, *see* inheritance

deserialization, 280

design
classes
encapsulation, 380
form classes, 382
generic classes, 381-382
inheritance, 380-381
MFC, 382
DLLs, 409
menus
context menus, 116
Windows conventions, 107

destructors, 564

Developer Studio
editor area, 9
Output pane, 9
toolbars, 10
workspace, 8

device contexts (graphics), 150-151
bitmaps, 154-156
brushes
creating, 153
patterns, 154
selecting, 153
device contexts class, 151
modifying, 656
pens
colors, 152-153
creating, 151
styles, 152
printing, 643-644

DevMode structure, 652

dialog windows
ActiveX controls, adding, 186-187
Add Member Function, 18
Add Member Variable, 32
Add Property, 475
Adding a Class, 113
breakpoints, 690
common dialogs
adding to applications, 90-92
classes, 90
OnFileopen function, 91
creating, 15-17
custom dialogs, 93-96
calling, 98
header files, 96-98
OnBcustomdialog function, 98-99
OnBwhichoption function, 99
reading variables, 99-100
Debug Assertion Failed, 687
defined, 83
exception handling, 674
Graphics application
canvas dialog, 160-162
main dialog, 157-160

icons, 20-21
Insert Control, 491
Insert Resource, 109
Invoke Methods, 492
menus
attaching, 112
cascading, 106
context menus, 106
creating, 109-111
event handling, 112-115
hotkeys, 107
pull-down menus, 106
Windows conventions, 107
message boxes
button combinations, 85
icons, 85
MessageBox function, 84
OnYesnocancel functions, 88-89
sample application, 86-88
Message Options, 695
ODBC Microsoft Access Setup, 323
Print
accessing directly, 657-659
bypassing, 649
Collate check box, 651
Project Settings, 678
Select Database Tables, 325

directives
#import, 349-350
#include directives, 97
CModArt, 392
CModArt class, 392
CTestAppDoc, 394, 414
DbAdo application, 364
Graphics application, 162
Hello, World application, 545
SDI application, 213
Sock application, 510
Squiggle control, 481

DISABLENOSCROLL combo box style, 264

disabling, *see* turning on/off

Disassembly toolbar (debugger), 694

displaying
bitmaps, 174-176
current records, 306-307
current URL, 529-530
drawings
Draw function, 390
GetDrawing function, 425
ModArtDraw function, 421
ModArtNewDrawing function, 418
message boxes, 37-38
processes, 697
threads, 697
toolbars, 254
OnUpdateViewColorbar function, 255
OnViewColorbar function, 256
ShowControlBar function, 255
ShowWindow function, 255

DLLs (dynamic link libraries), 406-407
ADO, importing, 349-350
advantages, 406
designing, 409
MFC DLLs
creating, 410-413
editing, 414-416
export operations, 408
test application, 413-414
non-Visual C++ applications, 427-428
regular DLLs
adapting functions, 421-422
creating, 416-420
DEF files, 408
export operations, 408-409
module definition files, 422
testing, 423-426
see also library modules

DockControlBar function, 254

docking toolbars, 253-254

Document/View architecture, 200-201

documents
opening
OnNewDocument function, 303-304
OnOpenDocument function, 304-305
printing, 647-650

DoDataExchange function, 330

DoModal function, 90

DoPreparePrinting function, 648

DoSort function, 192

DoSpin function, 455

double data type, 548

Draw function, 390, 446-447

drawing areas
finding size of, 452
setting, 386

drawings
circles, 168-171
clearing, 390-391
ClearDrawing function, 391
ModArtClearDrawing function, 421

creating
Draw function, 446-447
ModArtDraw function, 421
ModArtNewDrawing function, 418
NewDrawing function, 389, 398-399
OnNewDocument function, 395
deleting, 396
DeleteContents function, 425
SDI application, 214
displaying
Draw function, 389-390
GetDrawing function, 396, 425
OnDraw function, 397, 426
lines
colors, 164-165
DrawLine function, 165-166
NewLine function, 386-388, 399-401, 411-416, 419-420
loading, 488-491
random, 388-389
rectangles, 168-171
rendering, 211-212
saving, 215-217, 395
serializing, 390
start/end points, 448-449
viewpoints, 448
see also color; graphics

DrawLine function, 165-166

DrawRegion function, 168-171

drop-down list boxes, 27, 264

DROPDOWN combo box style, 263

DROPDOWNLIST combo box style, 264

dynamic link libraries, *see* DLLs

dynamic record sets, 318

dynasets, 318

E

edit boxes, 26
disabling, 39-41
hiding, 39-41
text
clearing, 38-39
displaying, 37-38
see also message boxes

Edit function, 320

Edit menu
design conventions, 107
Insert New Control command, 491

editing
About window, 23
message box text, 22
recordsets, 320-321

editor area (Developer Studio), 9

Empty function, 725

EnableDocking function, 253

encapsulation, 380

EndDoc function, 659

EndPage function, 660

EndWaitCursor function, 63

END_ADO_BINDING macro, 354

enum statement, 571

EnumFontFamiliesEx function, 122-123

EnumFontFamProc function, 124

ENUMLOGFONTEX structure, 124

equal sign (=), 43, 549

Error object, 345

errors, *see* exceptions

escapement (fonts), 126

event handling
ActiveX controls, 193-194
combo boxes
CMainFrame message map, 267-268
OnSelChangeWidth function, 267
Graphics application
OnPaint function, 167, 171-172
OnRSelection function, 163-164
menus, 112
color menu, 235
COMMAND event, 114
context menus, 116
event-handler declarations, 233-234
message maps, 234-235
OnExit function, 114
OnHello function, 113-114
OnHelpAbout function, 114-115
SDI application, 209-211
sockets, 503-504
Squiggle control, 485-487

event messages, *see* messages

events
adding to ActiveX controls, 476
keyboard
GetKeyState function, 65
messages, 56-57
OnKeyDown function, 57-58, 61-62
virtual key codes, 65
mouse
drawing program, 49
messages, 48
OnLButtonDown function, 56

- OnMouseMove function, 49-50, 54-55
    - tracking, 53-55
  - thread synchronization, 439-440
  - timers
    - clock timer, 71-73
    - counting timer, 74-81
    - defined, 68
    - IDs, 70-71
    - interval range, 81
    - maximum of, 81

exceptions
- catching, 661-666
  - blanket exception catch, 665
  - catch keyword, 663-664
  - multiple exceptions, 664-665
  - try keyword, 207, 663-664
- deleting, 668
- defined, 661
- MFC
  - CArchiveException class, 671-672
  - CDAOException class, 672-673
  - CDBException class, 672-673
  - CException class, 668-669
  - CFileException class, 667, 670-671
  - CMemoryException class, 669
  - CNotSupportedException class, 674
  - COleDispatchException class, 673-674
  - COleException class, 673-674
  - CResourceException class, 670
  - CUserException class, 674
  - generating custom, 675-676
- sockets, 504
- testing exception type, 665
- throwing, 666-667
- *see also* debugging techniques

executing commands, 351

Exit buttons, creating, 36-37

exporting
- classes, 408
- functions, 408-409

extended properties, 474

extraction operator (>>), 548

F

fax machines, 645

Field object, 346

fields, accessing, 353-354

file extensions
- .cpp, 565
- .h, 565
- .hpp, 565
- .rc, 108

File menus, designing, 107

File View, 8

files
- DEF, 408
- header files, 565-566
  - library modules, 402
  - ModArtDll, 417
- resource files, 108

FillFontList function, 132-134

Find function, 727

FindIndex function, 704

FindOneOf function, 728

flags
- binary operations, 52-53
- defined, 52
- TPM_CENTERALIGN, 118
- TPM_LEFTALIGN, 118
- TPM_LEFTBUTTON, 118
- TPM_RIGHTBUTTON, 118

FLAT style toolbar control style, 250

flat-file databases
- application shell, 284-285
- clean-up functions, 302
- controls
  - properties, 286-287
  - variables, 287
- defined, 284
- documents
  - document class, 294-295
  - opening, 303-305
- position counters, 297-298
- record class header, 310
- records
  - adding, 295-297
  - counting, 297
  - displaying, 306-307
- recordsets
  - navigating, 298-301, 307-308
  - serializing, 301-302
- saving changes, 308-309, 311-312
- serializable classes, 288-289
  - constructors, 290
  - creating, 291-294
  - inline methods, 289-290
- views, 285
  - view class, 305-306
  - resetting, 309-310

FlexGrid control
- cell IDs, 189
- event handling, 193-194
- loading data into, 188-189
- random data, generating, 189-191
- sorting, 192-193

float data type, 548
fonts
CreateFont function
alternatives to, 139
bItalic argument, 127
bUnderline argument, 127
cStrikeOut argument, 127
declaring, 125
example, 125
lpszFacename argument, 129
nCharSet argument, 127-128
nClipPrecision argument, 128
nEscapement argument, 126
nHeight argument, 126
nOrientation argument, 126
nOutPrecision argument, 128
nPitchAndFamily argument, 129
nQuality argument, 128
nWeight argument, 126-127
nWidth argument, 126
listing available fonts
callback functions, 122
EnumFontFamiliesEx function, 123
EnumFontFamProc function, 124
TrueType fonts, 140
sample application
application shell, 129-131
font list, 131-132
OnChangeEsamptext function, 136
OnInitDialog function, 133-135
OnSelchangeLfonts function, 138
SetMyFont function, 137
form classes, 382
Format function, 720
formatting
dates/time, 720-721
strings, 728-729
forms, populating, 367-369
Forward button (browser application), 531
forwardOnly recordsets, 318
frames, attaching toolbars to, 247-249
frameworks (MDI/SDI), printing, 637
friend classes, 565
friend functions, 565
functions, 546
AbortDoc, 657
Accept, 501
Add, 701
AddHead, 703
AddLine, 206
AddNew, 320, 357, 374
AddNewRecord, 296
AddString, 264
AddTail, 703
AfxBeginThread, 432-433, 468
AfxGetStaticModuleState, 409
BeginWaitCursor, 63
BitBlt, 155
BottomRight, 712
CalcPoint, 451
callback functions, 72, 122
CenterPoint, 713
ClearDrawing, 391
Close, 319, 357, 503
CoInitialize, 350
COMMAND, 227
CommandToIndex, 252
Compare, 726
CompareNoCase, 726
Connect, 500
constructors, 564
CoUninitialize, 350
CPrintDialog class, 653
CPrintInfo, 642
CreateBlankRecord, 374-375
CreateColorbar, 261-262
CreateEx, 250-251
CreateFont
alternatives to, 139
bItalic argument, 127
bUnderline argument, 127
cStrikeOut argument, 127
declaring, 125
example, 125
lpszFacename argument, 129
nCharSet argument, 127-128
nClipPrecision argument, 128
nEscapement argument, 126
nHeight argument, 126
nOrientation argument, 126
nOutPrecision argument, 128
nPitchAndFamily argument, 129
nQuality argument, 128
nWeight argument, 126-127
nWidth argument, 126
CreateHatchBrush, 655
CreateInstance, 350
CreateProcess, 44

declaring, 546
default access permissions, 292
DeflateRect, 715
Delete, 320-321, 376, 668
DeleteContents, 214, 302, 367, 396, 425
destructors, 564
DockControlBar, 254
DoDataExchange, 330
DoModal, 90
DoPreparePrinting, 648
DoSort, 192
DoSpin, 455
Draw, 390, 446-447
DrawLine, 165-166
DrawRegion, 168-171
Edit, 320
Empty, 725
EnableDocking, 253
EndDoc, 659
EndPage, 660
EndWaitCursor, 63
EnumFontFamiliesEx, 122-123
EnumFontFamProc, 124
exporting, 408-409
FillFontList, 132-134
Find, 727
FindIndex, 704
FindOneOf, 728
Format, 720
friend functions, 565
GenerateError, 365
GenID, 189
GetAt, 701, 704
GetBitmap, 174
GetBusy, 524
GetClientRect, 452
GetCopies, 653
GetCount, 704
GetCurRecord, 298
GetCurRecordNbr, 298
GetCurrentTime, 720
GetDay, 720
GetDayOfWeek, 720
GetDayOfYear, 720
GetDC, 448
GetDeviceCaps, 645
GetDlgItem, 39-41
GetDocument, 211
GetDrawing, 397, 425
GetDriverName, 653
GetErrorMessage, 668-669
GetFirstPosition, 706
GetFirstRecord, 299
GetFirstViewPosition, 270, 303
GetFixedRows, 188
GetFromPage, 653
GetHour, 720
GetItemRect, 263
GetKeyState, 65
GetLastError, 504
GetLastRecord, 301
GetLength, 479, 725
GetLine, 208
GetLineCount, 208
GetLocationURL, 524, 529
GetMaxID, 332
GetMinute, 720
GetMonth, 720
GetMouseCol, 194
GetMouseRow, 194
GetNext, 703
GetNextAssoc, 706
GetNextRecord, 299
GetNextView, 270, 303
GetPortName, 653
GetPrev, 703
GetPrevRecord, 300
GetPrinterDC, 653
GetRecordCount, 319
GetRecSe, 364
GetRows, 188
GetSecond, 720
GetSize, 701
GetSquiggleLength, 482
GetStatus, 720
GetToPage, 653
GetTopLevelFrame, 270
GetTotalDays, 723
GetTotalHours, 723
GetTotalMinutes, 723
GetTotalRecords, 297
GetTotalSeconds, 723
GetUpperBound, 701
GetViewWnd, 451
GetWindowText, 528
GetYear, 720
GoBack, 524
GoForward, 524
GoHome, 524
GoSearch, 524
Height, 713
InflateRect, 715
InitSpinners, 453
inline functions, 289
InsertAfter, 703
InsertAt, 702
InsertBefore, 703
InterSectRect, 713
Invalidate, 163
IsBOF, 319
IsEmpty, 725
IsEOF, 319
IsKindOf, 665
IsPrinting, 643
IsRectEmpty, 713
IsRectNull, 713
KillTimer, 79
Left, 727
Listen, 501
LoadBitmap, 172
LoadCursor, 59
LoadData, 188
LoadDrawing, 488-491
LoadStandardCursor, 59
LoadString, 264
LoadToolBar, 250
Lock, 439

Lookup, 705
main (Hello, World application), 545-546
MakeLower, 727
MakeReverse, 727
MakeUpper, 727
MessageBox, 22
  arguments, 84-85
  return value IDs, 85-86
Mid, 727
ModArtClearDrawing, 421
ModArtDraw, 421
ModArtInit, 428
ModArtNewDrawing, 427
Move, 319
MoveFirst, 319, 372
MoveLast, 319
MoveNext, 319
MovePrev, 319
Navigate, 524
NewDataSet, 310
NewDrawing, 398, 480
  library module, 389
  regular DLL, 418
NewLine, 399-401, 480
  library module, 386-388
  MFC DLL, 411-416
  regular DLL, 419-420
NormalizeRect, 717
Offset, 712
OnAccept, 504, 510, 513
OnBbitmap, 173-174
OnBclose, 518
OnBconnect, 512
OnBcustomdialog, 98-99
OnBeginPrinting, 655
OnBfirst, 307
OnBlast, 308
OnBsend, 515-516
OnBwhichoption, 99
OnCbemployed, 309, 312
OnCbonidle, 456
OnCbthread1, 464
OnCbthread2, 465
OnChangeInterval, 75
OnChangeEsamptext, 136
OnClick, 487
OnClose, 504, 517-518
OnClrmsg, 38-39
OnConnect, 504, 513
OnContextMenu, 117, 238-239
OnCreate, 248-249, 265-266
OnDataFirst, 371
OnDestroy, 467
OnDocumentComplete, 529
OnDraw, 397, 426, 638-641
  CSquiggleCtrl class, 486
  SDI application, 212
OnEndPrinting, 655
OnExit, 36
OnFileopen, 91, 103
OnGoBack, 531
OnGoNext, 531
OnHello, 19, 114
OnHelpAbout, 114-115
OnIdle, 431, 457-460
OnInitDialog, 34-35, 511
  ActiveX application, 192-193
  clock timer, 71-72
  counting timer, 76
  Fonts application, 133-135
  Graphics application, 161-162
  Mouse drawing program, 60-61
OnKeyDown, 57-58, 61-62
OnLButtonDown, 56
OnMouseDownMsfgrid, 194
OnMouseMove, 49-50, 54-55
OnMouseUpMsfgrid, 195
OnNewAddress, 527
OnNewDocument, 303, 366, 395, 424, 469
  SDI application, 221
  Spinner application, 454-455
OnOK, 36
OnOpenDocument, 304
OnPaint, 167, 171-172, 176, 714
OnPreparePrinting, 648
OnPrint, 638, 641-643
OnReceive, 504, 516
OnRecordDelete, 334
OnRecordNew, 333
OnRSelection, 164
OnRType, 508
OnRunpgm, 42-43
OnSelchangeLfonts, 138
OnSelChangeWidth, 267
OnSend, 504
OnSetCursor, 62-63
OnShwmsg, 37-38
OnStarttime, 76-77
OnStoptimer, 77
OnTimer
  clock timer, 73
  counting timer, 77-79
OnUpdateIndicatorColor, 273-274
OnUpdateViewColorbar, 255
OnViewColorbar, 256
OnViewRefresh, 534
OnViewStop, 533
OnYesnocancel, 88-89
Open, 318
overloading, 550-552
ParseDateTime, 722
PopulateView, 306
PrintAll, 653
PrintCollate, 653
PrintRange, 653
PrintSelection, 653

prototypes, 546
  declaring, 548
  example, 547
PtInRect, 715
PulseEvent, 440
rand, 388
RandomStringValue, 190-191
RecalcLayout, 256
Receive, 502
Refresh, 524, 533
RefreshBoundData, 368-369
RemoveAll, 302, 702
RemoveAt, 702-704
ReportError, 669
Requery, 320-321
ResetEvent, 440
ResumeThread, 469
ReverseFind, 728
RGB, 51
Right, 727
SelectObject, 219
Send, 502
Serialize, 281, 396
  CModArt class, 390
  declaring, 283
  example, 281, 291
  SDI application, 215-216, 219
SetAbsolutePosition, 319
SetAddress, 530
SetAt, 701, 704
SetButtonInfo, 262
SetButtonStyle, 252
SetButtonText, 276
SetCapture, 211
SetCheck, 225, 255
SetColPosition, 195
SetColSel, 192
SetCurSel, 269
SetCursor, 59
SetDate, 720
SetDateTime, 720
SetDateTimeSpan, 723
SetEvent, 440
SetIndicators, 272
SetLength, 452, 479
SetMapMode, 156, 448
SetMergeCol, 189
SetMyFont, 137-140
SetPaneText, 277
SetParent, 509
SetPixel, 50
SetPoint, 452
SetRect, 386, 713
SetRectEmpty, 713
SetRedraw, 195
SetSize, 701
SetSort, 192
SetSquiggleLength, 482
SetStatus, 720
SetTextArray, 189
SetTime, 720
SetTimer, 72
SetViewportOrg, 448
SetWindowText, 277
ShellExecute, 44
ShowBitmap, 174-175
ShowControlBar, 255
ShowWindow, 255
Size, 713
StartDoc, 659
StartPage, 660
Stop, 524, 532
strcpy, 726
StretchBlt, 155
strlen, 726
SubtractRect, 714
SuspendSpinner, 462-463, 470-471
SuspendThread, 469
ThreadFunc, 461, 470
ThrowErrno, 671
ThrowOsError, 671
TopLeft, 712
TrackPopupMenu, 117
TrimLeft, 727
TrimRight, 727
UnionRect, 713
Unlock, 439-440
Update, 320, 356
UpdateBoundData, 370-371
UpdateColorbar, 269-270
UpdateData, 35, 456
UpdateWidthCB, 269
WaitForSingleObject, 463
Width, 713
WinExec, 44

G

g prefix, 554
GDI objects, printing, 655
GenerateError function, 365
generating random data, 189-191
generic classes, 381-382
GenID function, 189
GetAt function, 701, 704
GetBitmap function, 174
GetBusy function, 524
GetClientRect function, 452
GetCopies function, 653
GetCount function, 704
GetCurRecord function, 298
GetCurRecordNbr function, 298
GetCurrentTime function, 720
GetDay function, 720
GetDayOfWeek function, 720
GetDayOfYear function, 720
GetDC function, 448
GetDeviceCaps function, 645
GetDlgItem function, 39-41
GetDocument function, 211
GetDrawing function, 397, 425
GetDriverName function, 653
GetErrorMessage function, 668-669
GetFirstPosition function, 706

GetFirstRecord function, 299
GetFirstViewPosition function, 270, 303
GetFixedRows function, 188
GetFromPage function, 653
GetHour function, 720
GetItemRect function, 263
GetKeyState function, 65
GetLastError function, 504
GetLastRecord function, 301
GetLength function, 479, 725
GetLine function, 208
GetLineCount function, 208
GetLocationURL function, 524, 529
GetMaxID function, 332
GetMinute function, 720
GetMonth function, 720
GetMouseCol function, 194
GetMouseRow function, 194
GetNext function, 703
GetNextAssoc function, 706
GetNextRecord function, 299
GetNextView function, 270, 303
GetPortName function, 653
GetPrev function, 703
GetPrevRecord function, 300
GetPrinterDC function, 653
GetRecordCount function, 319
GetRecSe function, 364
GetRows function, 188
GetSecond function, 720
GetSize function, 701
GetSquiggleLength function, 482
GetStatus function, 720
GetToPage function, 653
GetTopLevelFrame function, 270
GetTotalDays function, 723
GetTotalHours function, 723
GetTotalMinutes function, 723
GetTotalRecords function, 297
GetTotalSeconds function, 723
GetUpperBound function, 701
GetViewWnd function, 451
GetWindowText function, 528
GetYear function, 720
global variables
example, 552-553
naming conventions, 554
GoBack function, 524
GoForward function, 524
GoHome function, 524
GoSearch function, 524
graphical user interfaces, *see* GUIs
graphics, 150
bitmaps
copying, 155
displaying, 174-176
loading, 155, 172-174
resizing, 155
variables, 173
color
brushes, 153
CModArt class color table, 392
menus, 235, 240
pens, 152-153
SDI application, 218-222
specifying, 51
Spinner application, 445-446
coordinate-handling classes, 710
CPoint class, 712
CRect class, 712-717
Offset function, 712
operator overloads, 711-712, 716
device contexts, 150-151
bitmaps, 154-156
brushes, 153-154
device contexts class, 151
pens, 151-153
drawings, *see* drawings
icons, 20-21
mapping modes, 156
printing, 638-641
see also GUIs
grid control, *see* FlexGrid control
GUIs (graphical user interfaces)
combo boxes
creating, 260-266
event handling, 267-268
placeholders, 262
populating, 264
positioning, 263
resource files, 257-260
sizing, 276
styles, 263-264
updating, 268-270
menus, *see* menus
status bars
adding panes, 271-272
messages, 273-275
toolbars, 244-245
attaching to application frames, 247-249
borders, 251
buttons, 252-253, 276
child window IDs, 251
control styles, 250
creating, 246, 250
docking, 253-254
hiding, 254-256
loading, 250
naming, 247
styles, 250-251

H

.h file extension, 565
hash keys, troubleshooting, 710
HASSTRINGS combo box style, 264
header files, 565-566
custom dialog windows, 96-98
library modules, 402
ModArtDll, 417
height
fonts, 126
rectangles, 713
Height function, 713
Hello application
application framework, 12-14
dialog box icon, 20-21
project workspace, 11-12
source code, 17-19
window layout
designing, 15-17
maximize/minimize buttons, 21-22
Hello, World application
comments, 545
main function, 545-546
preprocessor directives, 545
return statement, 546
running, 543
source code listing, 543
workspace, 541-543
Help menus, designing, 107
helper classes (MFC)
collection classes, 699-700
array classes, 700-702
customizing, 707-710
list classes, 702-704
map classes, 704-706
type-safety, 707-710
coordinate-handling classes, 710
CPoint class, 712
CRect class, 712-717
CSize class, 717-718
Offset function, 712
operator overloads, 711-712, 716-718
string manipulation class
adding text, 725
case sensitivity, 727
constructor types, 724
formatting, 728-729
manipulation functions, 727
NULL-terminated types, 726
searches, 727-728
size, 725
time-handling classes, 718-719
COleDateTime class, 719-722
COleDateTimeSpan class, 722-724
formatting code, 720-721
hiding
edit boxes, 39-41
toolbars, 254
OnUpdateViewColorbar function, 255
OnViewColorbar function, 256
ShowControlBar function, 255
ShowWindow function, 255
HIENGLISH mapping mode, 156
HIGHEST thread priority, 432
HIMETRIC mapping mode, 156
hit-testing rectangles, 715-716
hotkeys, 107
.hpp file extension, 565
HTML (Hypertext Markup Language) viewer
browser controls, 524
browser status, 524
navigation functions, 523-524

I

icons
button icons, 246
creating, 20-21
message boxes, 85
IDispatch interface, 181
idle processing threads, 431
building, 457-459
continuous tasks, 459
OnIdle function, 455
starting, 456
stopping, 456
IDLE thread priority, 432
IDR_DAY11TYPE menu, 233
IDR_MAINFRAME menu, 233
IDs (identifiers)
controls, 44-45
message boxes, 85-86
timers, 70-71
if statement, 549
images, *see* graphics
IMPLEMENT_SERIAL macro, 282-283, 293
#import directive, 349-350
importing ADO DLLs, 349-350
#include directives, 97
CModArt, 392
CModArt class, 392
CTestAppDoc, 394, 414

DbAdo application, 364
Graphics application, 162
Hello, World application, 545
SDI application, 213
Sock application, 510
Squiggle control, 481

independent threads, 431, 460
arrays, 433
clean-up procedures, 466-468
creating, 432
main thread functions, 461
priorities, 432-433
stack size, 433
starting, 462-463
stopping, 462-463
suspended mode, 433
synchronizing, 434-435
critical sections, 436
events, 439-440
mutexes, 436-438
semaphores, 438-439
triggering from view object, 463-464

indexing map classes, 704-706
indirection operator (*), 554
InflateRect function, 715
inheritance, 380-381
Car class example, 572-576
Human class example, 569-572

initializing
controls, 33-35
printing, 654, 660

InitSpinners function, 453
inline functions, 289
see also functions

Insert Control dialog box, 491
Insert Resource dialog box, 109
InsertAfter function, 703
InsertAt function, 702
InsertBefore function, 703
inserting
records
ADO, 357, 373-374
ODBC, 320, 331-334
toolbars, 246

int data type, 548
interfaces, IDispatch, 181
Internet applications, 519
Internet Explorer
ActiveX document model, 522
HTML viewer
browser controls, 524
browser status, 524
navigation function, 523-524

intersection (rectangles), 713
InterSectRect function, 713
Invalidate function, 163
Invoke Methods dialog box, 492
IsBOF function, 319
IsEmpty function, 725
IsEOF function, 319
IsKindOf function, 665
ISOTROPIC mapping mode, 156
IsPrinting function, 643
IsRectEmpty function, 713
IsRectNull function, 713
italic text, formatting, 127

J-K

just-in-time debugging, 684
see also debugging techniques

KeepCurrentDrawing property (Squiggle control), 483
keyboard events
drawing program
cursor, 59
OnInitDialog function, 60-61
OnKeyDown function, 57-58, 61-62
OnSetCursor function, 62-63
GetKeyState function, 65
messages, 56-57
virtual key codes, 65

keyboard shortcuts, 107
keys
accelerators
creating, 118-119
defined, 107
restrictions, 119
Web Browser application, 532
hotkeys, 107
modifiers, 532

keywords, *see* statements
KillTimer function, 79

L

l prefix, 554
landscape orientation (printing), 654
Layout menu commands, Tab Order, 30
Left function, 727
library modules, 382-383
advantages, 402
creating, 383-385
header files, 402
static library project
class definitions, 385
clearing drawings, 390-391

creating drawings, 386-389
displaying drawings, 389-390
drawing area, 386
project module, 384-385
serializing drawings, 390
test application, 392-395
see also DLLs

licensing issues (ActiveX controls), 184, 196, 477

lines
adding, 206
CLine class
color, 218-219
constructor, 204-205
creating, 203
header file, 216
serialization, 217
counting, 207-208
drawing, 164
colors, 164-165
DrawLine function, 165-166
NewLine function, 386-388, 411-416, 419-420
retrieving, 208

list classes, 702-704

LIST toolbar control style, 250

Listen function, 501

listening for messages, 501

listing available fonts
callback functions, 122
EnumFontFamiliesEx function, 123
EnumFontFamProc function, 124
example, 131-133
FillFontList function, 132
TrueType fonts, 140

listings
ActiveX application
DoSort function, 192
GenID function, 189
LoadData function, 188
OnInitDialog function, 192-193
OnMouseDownMsfgrid function, 194
OnMouseUpMsfgrid function, 195
RandomStringValue function, 190-191
catching exceptions
all types of exceptions, 665
try…catch block, 663
try block with function calls, 664
two different exceptions, 664-665
CCustomException class, 675-676
Clasfarm program
.cpp file, 566
header file, 565
classes
declarations, 562-563
nesting, 566-567
clock timer
OnInitDialog function, 71-72
OnTimer function, 73
CModArt class
constructor, 479
GetLength function, 479
NewDrawing function, 480
NewLine function, 480
SetLength function, 479
CModArt library module
#include directives, 392
ClearDrawing function, 391
color table, 392
constructor, 391
Draw function, 390
NewDrawing function, 389, 398
NewLine function, 386, 399
Serialize function, 390
SetRect function, 386
combo boxes
CMainFrame message map, 268
CreateColorBar function, 261-262
OnCreate function, 265-266
OnSelChangeWidth function, 267
OnUpdateViewColorbar function, 255
OnViewColorbar function, 256
project resource files, 258
UpdateColorBar function, 269-270
UpdateWidthCB function, 269
Width menu command event handler, 270
counting timer
OnInitDialog function, 76
OnStarttime function, 76-77
OnStoptimer function, 77
OnTimer function, 77-78
Stop button, 80-81
CTestAppDoc class
DeleteContents function, 396
GetDrawing function, 397

#include directives, 394
OnNewDocument function, 395
Serialize function, 396
CTestAppDoc class
declaration, 423-424
DeleteContents function, 425
GetDrawing function, 425
#include directives, 414
OnNewDocument function, 424
Serialize function, 425
CTestAppView class, OnDraw function, 397, 426
DbAdo database
AddNew function, 374
CreateBlankRecord function, 374-375
custom record class, 362-363
Delete function, 376
DeleteContents function, 367
GenerateError function, 365
GetRecSet function, 364
#include directives, 364
MoveFirst function, 372
OnDataFirst function, 371
OnNewDocument function, 366
RefreshBoundData function, 368-369
UpdateBoundData function, 370-371
DbOdbc database
CDbOdbcSet constructor, 331
Database field variables, 329
DoDataExchange function, 330
GetMaxID function, 332
OnRecordDelete function, 334
OnRecordNew function, 333
dialog windows
header files, 97
OnBcustomdialog function, 98-99
OnBwhichoption function, 99
OnFileopen function, 91, 103
OnYesnocancel function, 88-89
OPENFILENAME structure, 102
flat-file database
AddNewRecord function, 296-297
CPerson class declaration, 292
CPerson constructor, 290
custom object header, 305-306
DeleteContents function, 302
document class implementation, 295
Get/Set inline functions, 290
GetCurRecord function, 298
GetCurRecordNbr function, 298
GetFirstRecord function, 299
GetLastRecord function, 301
GetNextRecord function, 299-300
GetPrevRecord function, 300
GetTotalRecords function, 297
IMPLEMENT_SERIAL macro, 293-294
NewDataSet function, 310
OnBfirst function, 307-308
OnBlast function, 308
OnCbemployed function, 309, 312
OnNewDocument function, 303-304
OnOpenDocument function, 304-305
PopulateView function, 306-307
record class header, 310
Serialize function, 291, 302
Fonts application
FillFontList function, 132-134
listing available fonts, 131-132
OnChangeEsamptext function, 136
OnInitDialog function, 133-135
OnSelchangeLfonts function, 138
SetMyFont function, 137-140
function prototype, 547
GetDlgItem function, 39-41
global/local variables, 552-553
Graphics application
color table, 164-165
DrawLine function, 165-166

DrawRegion function, 169-171
#include directives, 162
OnBbitmap function, 173-174
OnInitDialog function, 161-162
OnPaint function, 167, 171-172, 176
OnRSelection function, 164
ShowBitmap function, 175
Hello application, 19
Hello, World application, 543
inheritance
Allauto class example, 575
Auto class example, 573-574
Car class example, 574-575
Human class example, 570-571
MDI application
color menu event handlers, 235
context menu, 238-239
event-handler declarations, 233-234
message map entries, 234-235
menu functions
OnContextMenu function, 117
OnHello function, 114
OnHelpAbout function, 114-115
ModArtDll
CModArt class definition, 411
header file, 417
ModArtClearDrawing function, 421-422
ModArtDraw function, 421
ModArtInit function, 428
ModArtNewDrawing function, 418, 427
module definition file, 422
NewLine function, 412-416, 419-420
Serialize function, 421
Mouse drawing program
OnInitDialog function, 60-61
OnKeyDown function, 57-58, 61-62
OnLButtonDown function, 56
OnMouseMove function, 49-50, 54-55
OnSetCursor function, 62-63
OnClrmsg function, 38-39
OnExit function, 36
OnInitDialog function, 34-35
OnRunpgm function, 42-43
OnShwmsg function, 37-38
overloaded functions, 550-552
pointers, 555-557
raising exceptions, 666-667
SDI application
AddLine function, 206, 221-222
Black menu functions, 224
Blue menu functions, 224-225
CLine constructor, 204, 218
color table specification, 220
DeleteContents function, 214
Draw function, 205, 219
GetColor function, 222
GetLine function, 208
GetLineCount function, 208
#includes directives, 213
Line.cpp file, 217
Line.h file, 216
mouse functions, 209-210
OnDraw function, 212
OnNewDocument function, 221
Serialize function, 215-216, 219
serialization
DECLARE_SERIAL macro, 282
IMPLEMENT_SERIAL macro, 282-283
Serialize function, 281
Sock application
#include directives, 510
OnAccept function, 510, 513
OnBclose function, 518
OnBconnect function, 512
OnBsend function, 515-516
OnClose function, 517-518
OnConnect function, 513
OnInitDialog function, 511
OnReceive function, 516
OnRType function, 508
SetParent function, 509
Spinner application
CalcPoint function, 451
color table, 445-446
CSpinner class declaration, 445

CSpinner constructor, 444
CSpinner source file, 449
CTaskingDoc source file, 450
DoSpin function, 455
Draw function, 446-447
InitSpinners function, 453
OnCbonidle function, 456
OnCbthread1 function, 464
OnCbthread2 function, 465
OnDestroy function, 467
OnIdle function, 457-460
OnNewDocument function, 454-455, 469
SuspendSpinner function, 462-463, 470
ThreadFunc function, 461, 470

Squiggle control
constructor, 485
GetSquiggleLength function, 482
#include directives, 481
LoadDrawing function, 488-491
OnClick function, 487
OnDraw function, 486
SetSquiggleLength function, 482

status bars
CToolbarDoc message map, 274
indicator table definition, 272
OnUpdateIndicatorColor function, 273-274

structures, 560
toolbars, attaching to application frames, 248-249
TRACE macro example, 686-687
WebBrowse application
CMainFrame message map, 528
OnDocumentComplete function, 529
OnGoBack function, 531
OnGoNext function, 531
OnNewAddress function, 527
OnViewRefresh function, 534
OnViewStop function, 533
SetAddress function, 530

LoadBitmap function, 172
LoadCursor function, 59
LoadData function, 188
LoadDrawing function, 488-491
loading data
bitmaps, 155
LoadBitmap function, 172
LoadImage function, 172
OnBbitmap function, 173-174
cursors into memory, 59
drawings, 488-491
FlexGrid control, 188-189
toolbars, 250
LoadStandardCursor function, 59
LoadString function, 264
LoadToolBar function, 250
local variables
example, 552-553
naming conventions, 554

Lock function, 439
locking semaphores, 439
LOENGLISH mapping mode, 156
LOGFONT structure, 123
logical operators, 550
AND (&&), 51
OR (||), 52
LOMETRIC mapping mode, 156
long int data type, 548
Lookup function, 705
loopback computer name, 511
LOWEST thread priority, 432
lpqzFacename argument (CreateFont function), 129

M

macros
ADO_FIXED_LENGTH_ENTRY, 355
ADO_NUMERIC_ENTRY, 355
ADO_VARIABLE_LENGTH_ENTRY, 355-356
AFX_EXT_CLASS, 408
AFX_MANAGE_STATE, 409
ASSERT, 687-689
BEGIN_ADO_BINDING, 354
DECLARE_SERIAL, 282, 292
END_ADO_BINDING, 354
IMPLEMENT_SERIAL, 282-283, 293
TRACE, 685-687, 708
VERIFY, 687-689
main function (Hello, World application), 545-546

MakeLower function, 727
MakeReverse function, 727
MakeUpper function, 727
map classes, 704-706
mapping modes, 156, 448
mathematical operators, 549
maximize/minimize buttons, 21-22
MB_ABORTRETRYIGNORE button ID, 85
MB_OK button ID, 85
MB_OKCANCEL button ID, 85
MB_RETRYCANCEL button ID, 85
MB_YESNO button ID, 85
MB_YESNOCANCEL button ID, 85
MDI (Multiple Document Interface) applications, 229-230
- application shell, 231-232
- classes, 230-231
- context menus, 236-239
- drawing functionality, 232
- menu handling
  - color menu, 235
  - event-handler declarations, 233-234
  - message maps, 234-235
- SDI applications, compared, 239

Member Variables tab (Class Wizard), 32
Memory window (debugger), 694
Menu Designer
- creating menus
  - captions, 110
  - properties, 110
  - separators, 111
- starting, 109

menus
- accelerators
  - creating, 118-119
  - restrictions, 119
- attaching, 112
- captions, 110
- cascading, 106
- context menus, 106
  - creating, 116-118, 236-239
  - defined, 106
  - designing, 116
  - event handling, 116
- designing, 107
- event handling, 112
  - color menu, 235
  - COMMAND event, 114
  - event-handler declarations, 233-234
  - message maps, 234-235
  - OnExit function, 114
  - OnHello function, 113-114
  - OnHelpAbout function, 114-115
- hotkeys, 107
- IDR_DAY11TYPE, 233
- IDR_MAINFRAME, 233
- items, adding, 110
- menu resources, creating, 109
- naming, 119
- properties, 110
- SDI application
  - Black menu functions, 224
  - Blue menu functions, 224-225
  - creating, 222
  - properties, 223
- separator lines, 111
- troubleshooting, 240

Menus application
- application shell, 108-109
- attaching menus, 112
- creating menus
  - menu items, 110
  - menu resources, 109
  - separator lines, 111
- event handling, 112
  - OnExit function, 114
  - OnHello function, 113-114
  - OnHelpAbout function, 114-115

message boxes
- creating
  - button combinations, 85
  - icons, 85
  - MessageBox function, 84
- MessageBox function, 85-86
- sample application
  - control properties, 86-87
  - control variables, 88
  - OnYesnocancel functions, 88-89
- text, changing, 22

message maps
- CMainFrame, 268
- CToolbarDoc class, 274
- MDI application, 234-235
- ON_COMMAND_RANGE, 236

Message Maps tab (Class Wizard), 34
Message Options dialog box, 695
MessageBox function, 22
- arguments, 84-85
- return value IDs, 85-86

messages
- clearing, 38-39
- disabling, 39-41

displaying, 37-38
hiding, 39-41
keyboard events, 56-57
listening for, 501
mouse events, 48
receiving, 502-503, 516-517
sending, 502, 515-516
status bars, 273-275
windows, viewing, 695-696

methods, *see* functions

MFC (Microsoft Foundation Classes), 382, 699-700
array classes, 700-702
coordinate-handling classes, 710
CPoint class, 712
CRect class, 712-717
CSize class, 717-718
Offset function, 712
operator overloads, 711-712, 716-718
customizing, 675-676, 707-710
exception classes
CArchiveException class, 671-672
CDAOException class, 672-673
CDBException class, 672-673
CException class, 668-669
CFileException class, 667, 670-671
CMemoryException class, 669
CNotSupportedException class, 674
COleDispatchException class, 673-674
COleException class, 673-674
CResourceException class, 670
CUserException class, 674
form classes, 382
list classes, 702-704
map classes, 704-706
string manipulation class
adding text, 725
case sensitivity, 727
constructor types, 724
formatting, 728-729
manipulation functions, 727
NULL-terminated types, 726
searches, 727-728
size, 725
time-handling classes, 718-719
COleDateTime class, 719-722
COleDateTimeSpan class, 722-724
formatting code, 720-721
type-safety, 707-710

MFC extension DLLs
creating, 410
export operations, 408
ModArtDll project
class declaration, 411
NewLine function, 411-416
project file, 410
test application, 413-414
see also DLLs

MFC Tracer tool, 698

Microsoft Foundation Classes, *see* MFC

Mid function, 727

MM_ANSIOTROPIC mapping mode, 156

MM_HIENGLISH mapping mode, 156

MM_HIMETRIC mapping mode, 156

MM_ISOTROPIC mapping mode, 156

MM_LOENGLISH mapping mode, 156

MM_LOMETRIC mapping mode, 156

MM_TEXT mapping mode, 156

MM_TWIPS mapping mode, 156

mnemonics, 31

modal dialog windows, 92
see also dialog windows

ModArt application, 478-480
library module (random squiggle generator)
ClearDrawing function, 391
color table, 392
constructor, 391
creating, 385
Draw function, 390
#include directives, 392
NewDrawing function, 389, 397-398
NewLine function, 386, 399
rand function, 388
Serialize function, 390
SetRect function, 386
variables, 385
testing, 413-414
Squiggle control
CModArt class modifications, 478-481
control shell, 477-478
event handling, 485-487
events, 489-491
methods, 487-489
properties, 481-483
property page, 483-485
testing, 491-492

ModArtClearDrawing function, 421

ModArtDll application
MFC DLL
CModArt class declaration, 411
NewLine function, 411-416
project file, 410
testing, 413-414
regular DLL, 416
header file, 417
ModArtClearDrawing function, 421-422
ModArtDraw function, 421
ModArtNewDrawing function, 418
ModArtSerialize function, 421
module definition file, 422
NewLine function, 419-420
project file, 417
testing, 423-426
ModArtDraw function, 421
ModArtInit function, 428
ModArtNewDrawing function, 427
modeless dialog windows, 92
see also dialog windows
modifier keys (Web Browser application), 532
mouse events
handling
OnMouseDownMsfgrid function, 194
OnMouseUpMsfgrid function, 195
SDI application, 209-211
messages, 48
Mouse drawing program
application shell, 49
OnLButtonDown function, 56
OnMouseMove function, 49-50, 54-55
tracking mouse coordinates, 53-55
Move function, 319
MoveFirst function, 319, 372
MoveLast function, 319
MoveNext function, 319
MovePrev function, 319
moving toolbars, 10
Multiple Document Interface, *see* MDI applications
multiplication operator (*), 549
multitasking, 430
arrays, 433
clean-up procedures, 466-468
idle processing threads, 431
building, 457-459
continuous tasks, 459
OnIdle function, 455
starting, 456
stopping, 456
independent threads, 431, 460
creating, 432
main thread function, 461
priorities, 432-433
stack size, 433
starting, 462-463
stopping, 462-463
suspended mode, 433
trigging from view object, 463-464
thread synchronization, 434-435
critical sections, 436
events, 439-440
mutexes, 436-438
semaphores, 438-439
mutexes, 436-438
m_ prefix, 33
m_ofn property (CFileDialog class), 102

N

\n string, 247
naming conventions
constructors, 564
menus, 119
toolbars, 247
variables
global, 554
local, 554
m_ prefix, 33
Navigate function, 524
navigating
recordsets
ADO, 352-353
GetCurRecord function, 298
GetFirstRecord function, 299
GetLastRecord function, 301
GetNextRecord function, 299
GetPrevRecord function, 300
MoveFirst function, 372
ODBC, 319-320
OnBfirst function, 307
OnBlast function, 308
OnDataFirst function, 371
the Web, 523-524
nCharSet argument (CreateFont function), 127-128
nClipPrecision argument (CreateFont function), 128

nEscapement argument (CreateFont function), 126
nested classes, 566-569
networks, 496-497
debugging across, 685
Internet applications, 519
protocols, 497
sockets, 497
closing connections, 503, 517-519
controls, 505-508
creating, 498-500
error handling, 504
event handling, 503-504
listening for messages, 501
opening connections, 500-501, 511-514
ports, 498
receiving messages, 502-503, 516-517
sending messages, 502, 515-516
socket class, 509-511
new keyword, 666
NewDataSet function, 310
NewDrawing function, 398, 480
library module, 389
regular DLL, 418
NewLine function, 399-401, 480
library module, 386-388
MFC DLL, 411-416
regular DLL, 419-420
NEWTEXTMETRICEX structure, 124
nHeight argument (CreateFont function), 126
NOINTEGRALHEIGHT combo box style, 264
nOrientation argument (CreateFont function), 126
NORMAL thread priority, 432
NormalizeRect function, 717
nOutPrecision argument (CreateFont function), 128
nPitchAndFamily argument (CreateFont function), 129
nQuality argument (CreateFont function), 128
NULL-terminated strings, 726
NumberSquiggles property (Squiggle control), 483
nWeight argument (CreateFont function), 126-127
nWidth argument (CreateFont function), 126

O

object IDs (controls), 44-45
Object Linking and Embedding (OLE), 180
debugging objects, 684
exceptions, 673-674
see also ActiveX controls
object-oriented design
encapsulation, 380
inheritance, 380-381
Car class example, 572-576
Human class example, 569-572
objects
adding to classes, 708-709
ADO, *see* ADO
containers, 700-702
positioning, 450-452
ODBC (Open Database Connectivity), 316-317
Administrator, 322
data source configuration, 322-323
database connections, 317
exceptions, 672-673
records
adding, 331-334
deleting, 334-335
recordsets
closing, 319
editing, 320-321
navigating, 319-320
opening, 317-318
sample application
application shell, 324-325
CDbOdbcSet constructor, 331
control properties, 325-327
control variables, 328
Database field variables, 329
DoDataExchange function, 330
record management, 331-335
ODBC Microsoft Access Setup dialog box, 323
OEMCONVERT combo box style, 264
Offset function, 712
OLE (Object Linking and Embedding), 180
debugging objects, 684
exceptions, 673-674
see also ActiveX controls
OLE/COM Object Viewer tool, 697
OnAccept function, 504, 510, 513
OnBbitmap function, 173-174
OnBclose function, 518
OnBconnect function, 512
OnBcustomdialog function, 98-99
OnBeginPrinting function, 655

OnBfirst function, 307
OnBlast function, 308
OnBsend function, 515-516
OnBwhichoption function, 99
OnCbemployed function, 309, 312
OnCbonidle function, 456
OnCbthread1 function, 464
OnCbthread2 function, 465
OnChangeEsamptext function, 136
OnChangeInterval function, 75
OnClick function, 487
OnClose function, 504, 517-518
OnClrmsg function, 38-39
OnConnect function, 504, 513
OnContextMenu function, 117, 238-239
OnCreate function, 248-249, 265-266
OnDataFirst function, 371
OnDestroy function, 467
OnDocumentComplete function, 529
OnDraw function, 397, 426, 638-641
compared to OnPrint, 642
CSquiggleCtrl class, 486
SDI application, 212
OnEndPrinting function, 655
OnExit function, 36
OnFileopen function, 91, 103
OnGoBack function, 531
OnGoNext function, 531
OnHello function, 19, 114
OnHelpAbout function, 114-115
OnIdle function, 431, 457-460
OnInitDialog function, 34-35, 511
ActiveX application, 192-193
clock timer, 71-72
counting timer, 76
drawing program, 60-61
Fonts application, 133-135
Graphics application, 161-162
OnKeyDown function, 57-58, 61-62
OnLButtonDown function, 56
OnMouseDownMsfgrid function, 194
OnMouseMove function, 49-50, 54-55
OnMouseUpMsfgrid function, 195
OnNewAddress function, 527
OnNewDocument function, 303, 366, 395, 424, 469
SDI application, 221
Spinner application, 454-455
OnOK function, 36
OnOpenDocument function, 304
OnPaint function, 167, 171-172, 176, 714
OnPreparePrinting function, 648
OnPrint function, 638
compared to OnDraw, 642
overriding, 641-643
OnReceive function, 504, 516
OnRecordDelete function, 334
OnRecordNew function, 333
OnRSelection function, 164
OnRType function, 508
OnRunpgm function, 42-43
OnSelchangeLfonts function, 138
OnSelChangeWidth function, 267
OnSend function, 504
OnSetCursor function, 62-63
OnShwmsg function, 37-38
OnStarttime function, 76-77
OnStoptimer function, 77
OnTimer function
clock timer, 73
counting timer, 77-79
OnUpdateIndicatorColor function, 273-274
OnUpdateViewColorbar function, 255
OnViewColorbar function, 256
OnViewRefresh function, 534
OnViewStop function, 533
OnYesnocancel functions, 88-89
ON_COMMAND_RANGE message map, 236
Open Database Connectivity, *see* ODBC
Open function, 318
OPENFILENAME structure, 102
opening
database connections, 350-351
databases, 351-352
documents
OnNewDocument function, 303-304
OnOpenDocument function, 304-305
recordsets, 317-318
socket connections, 500-501, 511-514
operators
address-of (&), 554-555
assignment (=), 549
binary
AND (&), 51-52
attribute flags, 52-53
OR (|), 52
extraction operator (>>), 548

  - indirection (*), 554-556
  - logical, 550
    - AND (&&), 51
    - OR (||), 52
  - mathematical, 549
  - overloads, 711-712
    - COleDateTime class, 722
    - CRect class, 716
    - CSize class, 718
  - redirection (<<), 546
  - relational, 550
  - scope (::), 568
  - structure member operator (.), 561

optimizing printing, 656

OR operators
- binary OR (|), 52-53
- logical OR (||), 52

orientation (print jobs), 647, 654

Output pane (Developer Studio), 9

overloaded functions, 550-552

overriding OnPrint function, 641-643

OWNERDRAWFIXED combo box style, 264

OWNERDRAWVARIABLE combo box style, 264

P

page ranges, printing, 647-648

panes (status bars), adding, 271-272

Parameter object, 346

ParseDateTime function, 722

passing arguments by reference, 557-558

pens
- colors, 152-153
- creating, 151
- selecting, 64
- styles, 152

pipe character (|)
- binary OR (|), 52-53
- logical OR (||), 52

pitch (fonts), 129

pointers
- DbAdo database, 364
- declaring, 554
- defined, 211
- examples, 555-557
- Graphics application, 174
- Spinner application, 454

points
- hit-testing, 715-716
- intersection, 713

polymorphism, 550-552

pop-up menus, *see* context menus

PopulateView function, 306

populating
- combo boxes, 264
- forms, 367-369

portrait orientation (print jobs), 654

ports, 498

positioning objects, 450-452
- combo boxes, 263
- toolbars, 251

precompiler directives
- #import, 349-350
- #include directives, 97
  - CModArt, 392
  - CModArt class, 392
  - CTestAppDoc, 394, 414
  - DbAdo application, 364
  - Graphics application, 162
  - Hello, World application, 545
  - SDI application, 213
  - Sock application, 510
  - Squiggle control, 481

preemptive multitasking, *see* multitasking

prefixes, m_, 33

previewing
- AppWizard, 637
- support, 638

Print dialog box, 650-654
- accessing directly, 657-659
- bypassing, 649
- Collate check box, 651

PrintAll function, 653

PrintCollate function, 653

print jobs
- accessing directly, 657-660
- AppWizard, 637
- aspect ratio, 645-647
- canceling, 657
- coordinates
  - converting, 644
  - locating, 642
- device contexts, 643-644, 656
- DevMode structure, 652
- different display/printed output, 649
- GDI objects, 655
- graphics, 638-641
- initializing, 654, 660
- MDI framework, 637
- OnDraw function, 643
- optimizing, 656
- options, 652
- orientation, 647, 654
- page ranges, 648
- pagination, 647
- Print dialog box, 650-654
  - accessing directly, 657-659
  - bypassing, 649
  - Collate check box, 651

SDI framework, 637-641
spooling, 659
start/end pages, 647-650
status, checking, 660
support, 638
WYSIWYG, 641

PrintRange function, 653

PrintSelection function, 653

priority levels (threads), 432-433

Process Viewer tool, 697

processes, viewing, 697

programs, *see* applications

progress bars
messages, 273-275
panes, 271-272

Project Settings dialog box, 678

properties (ActiveX controls)
ambient, 474
custom, 474
extended, 474
property page, 483-485
specifying, 475
Squiggle control, 481-483
control settings, 484
control variables, 484
layout, 483

protocols, 497

prototypes (functions), 546
declaring, 548
example, 547

PtInRect function, 715

pull-down menus, 106
Black menu functions, 224
Blue menu functions, 224-225
creating, 222
properties, 223

PulseEvent function, 440

Q-R

QuickWatch window (debugger), 694

radio buttons, 27

rand function, 388

random data generation
FlexGrid control, 189-191
random squiggle generator
ClearDrawing function, 391
color table, 392
constructor, 391
creating, 385
Draw function, 390
#include directives, 392
NewDrawing function, 389, 397-398
NewLine function, 386, 399
rand function, 388
Serialize function, 390
SetRect function, 386
variables, 385

RandomStringValue function, 190-191

.rc filename extension, 108

readOnly flag (recordsets), 318

RecalcLayout function, 256

Receive function, 502

receiving messages, 502-503, 516-517

records
adding
AddNewRecord function, 295-297
ADO, 357, 373-374
ODBC, 320, 331-334
counting, 297
creating, 374-375
current record number, returning, 297-298
deleting
ADO, 357, 375-376
ODBC, 321, 334-335
displaying current, 306-307
fields, accessing, 353-354
saving, 308-312, 370-371
updating, 356

recordsets, 346
accessing
ADO, 353-354
ODBC, 319
closing, 357
editing, 320-321
navigating, 298
ADO, 352-353
GetCurRecord function, 298
GetFirstRecord function, 299
GetLastRecord function, 301
GetNextRecord function, 299
GetPrevRecord function, 300
MoveFirst function, 372
ODBC, 319-320
OnBfirst function, 307
OnBlast function, 308
OnDataFirst function, 371
opening, 317-318
types, 318

rectangles
center, 713
drawing, 168-171
height/width, 713
hit-testing, 715-716
intersection, 713

negative height/width, 717
sizing, 715
redirection operator (<<), 546
reference variables, 557-558
Refresh button (browser application), 533-535
Refresh function, 524, 533
RefreshBoundData function, 368-369
REGISTERDROP toolbar control style, 250
Registered ActiveX Controls folder, 184
registering ActiveX controls
Registered ActiveX Controls folder, 184-186
regsvr32 utility, 183
Registers window (debugger), 694
regsvr32 utility, 184
regular DLLs, *see* DLLs
relational operators, 550
Release mode, 678
remote debugging, 683-684
RemoveAll function, 302
RemoveAt function, 702-704
RemoverAll function, 702
ReportError function, 669
Requery function, 320-321
ResetEvent function, 440
resizing bitmaps, 155
Resource View, 8
resources
exception handling, 670
leaks, 151
resource files
defined, 108
editing, 257-260
ResumeThread function, 469
return statement, 546
return values
EnumFontFamProc function, 124
MessageBox function, 85-86
ReverseFind function, 728
RGB function, 51
Right function, 727
Run Program buttons, creating, 42-43
running applications, 543-544
runtime licenses, 477

S

saving
drawings, 215-217, 395
records, 308-312, 370-371
scope operator (::), 568
SDI (Single Document Interface) applications
classes
CDocument, 200
CFrameView, 200
CObArray class, 205
CView, 200
CWinApp, 200
color
AddLine function, 221-222
color table specification, 219-220
GetColor function, 222
lines, 218-219
OnNewDocument function, 220-221
DbOdbc database
adding records, 331-334
application shell, 324-325
CDbOdbcSet constructor, 331
control properties, 325-327
control variables, 328
Database field variables, 329
deleting records, 334-335
DoDataExchange function, 330
Document/View architecture, 200
drawings
deleting, 214
saving, 215-217
exception handling, 207
#includes directives, 213
lines
adding, 206
counting, 207-208
drawing, 203-205
retrieving, 208
MDI applications, compared, 239
mouse events, 209-211
printing, 637
pull-down menus
Black menu functions, 224
Blue menu functions, 224-225
creating, 222
properties, 223
rendering, 211-212
Select Database Tables dialog box, 325
selecting
brushes, 153
fonts, 136
OnSelchangeLfonts function, 138
SetMyFont function, 137
SelectObject function, 219
semaphores, 438-439
Send function, 502
sending messages, 502, 515-516
separators (menus), creating, 111

serialization
classes
CArchive, 280
CModArt, 390
CTestAppDoc, 396, 425
design, 283-284
defined, 279
macros, 281
DECLARE_SERIAL, 282
IMPLEMENT_SERIAL, 282-283
ModArtDll project, 421
Serialize function, 281, 396
CModArt class, 390
declaring, 283
examples, 281, 291
SDI application, 215-216, 219
Serialize application
application shell, 284-285
clean-up functions, 302
controls
properties, 286-287
variables, 287
CPerson class, 288
constructor, 290
inline functions, 289-290
serializing, 291-294
variables, 289
document class
implementation, 295
variables, 294
documents, opening, 303-305
record class header, 310
records
adding, 295-297
counting, 297
current record numbers, 297-298
displaying, 306-307
navigating, 298-301, 307-308
saving, 308-312
views, 285
resetting, 309-310
view class, 305-306
Serialize function, 281, 396
CModArt class, 390
declaring, 283
example, 281, 291
SDI application, 215-216, 219
SetAbsolutePosition function, 319
SetAddress function, 530
SetAt function, 701, 704
SetButtonInfo function, 262
SetButtonStyle function, 252
SetButtonText function, 276
SetCapture function, 211
SetCheck function, 225, 255
SetColPosition function, 195
SetColSel function, 192
SetCurSel function, 269
SetCursor function, 59
SetDate function, 720
SetDateTime function, 720
SetDateTimeSpan function, 723
SetEvent function, 440
SetIndicators function, 272
SetLength function, 452, 479
SetMapMode function, 156, 448
SetMergeCol function, 189
SetMyFont function, 137-140
SetPaneText function, 277
SetParent function, 509
SetPixel function, 50
SetPoint function, 452
SetRect function, 386, 713
SetRectEmpty function, 713
SetRedraw function, 195
SetSize function, 701
SetSort function, 192
SetSquiggleLength function, 482
SetStatus function, 720
SetTextArray function, 189
SetTime function, 720
SetTimer function, 72
SetViewportOrg function, 448
SetWindowText function, 277
shapes
center, 713
circles, 168-171
height/width, 713, 717
hit-testing, 715-716
intersection, 713
lines
colors, 164-165
DrawLine function, 165-166
NewLine function, 386-388, 399-401, 411-416, 419-420
rectangles, 168-171
sizing, 715
see also drawings; graphics
ShellExecute function, 44
short int data type, 548
ShowBitmap function, 174-175
ShowControlBar function, 255
ShowWindow function, 255
SIMPLE combo box style, 264
Single Document Interface, *see* SDI applications
single stepping (debugging)
breakpoints
setting, 689-690
toggling on/off, 690
step options, 691-692
Size function, 713
sizing
bitmaps, 155

combo boxes, 276
coordinate-handling classes
CSize class, 717-718
operator overloads, 718
rectangles, 715
strings, 725
toolbar borders, 251
slash command, 545
snapshot record sets, 318
Sock application (sockets)
connections
closing, 517-519
opening, 512-514
controls, 505
OnRType function, 508
properties, 506-507
variables, 508
#include directives, 510
initialization code, 511
messages
receiving, 516-517
sending, 515-516
shell, 505
socket class
member variables, 509
OnAccept function, 510
SetParent function, 509
sockets, 497
connections
closing, 503, 517-519
opening, 500-501, 511-514
creating, 498-500, 509-511
error handling, 504
event handling, 503-504
messages
listening for, 501
receiving, 502-503, 516-517
sending, 502, 515-516
ports, 498
see also Sock application
SORT combo box style, 264
sorting FlexGrid control, 192-193
Source Browser
base classes option, 682
Call Graph option, 683
Callers Graph option, 683
Definitions and References option, 681
derived classes option, 683
File Outline option, 682
identifiers, 681
Spinner application
framework
control properties, 441-442
control variables, 442
idle processing threads
building, 457-459
continuous tasks, 459
OnIdle function, 455
starting, 456
stopping, 456
independent threads, 460
Main thread function, 461
starting, 462-463
stopping, 462-463
triggering from view object, 463-464
shutdown procedures, 467
spinners
arrays, 449-450
class declaration, 444-445
color table, 445-446
designing, 443
drawing, 446-447
initializing, 452-455
positioning, 450-452
source file, 449
spinning, 455
start/end points, 448-449
variables, 443-444
viewpoint, 448
spooling print jobs, 659
Spy++ tool
Messages view, 695-696
Processes view, 697
Threads view, 697
Windows view, 696
Squiggle control
CModArt class modifications
constructor, 479
GetLength function, 479
member variables, 478
NewDrawing function, 480
NewLine function, 480
SetLength function, 479
control shell, 477-478
event handling, 485-487
events, 489-491
functions
class constructor, 485
LoadDrawing, 488-491
OnClick, 487
OnDraw, 486
#include directives, 481
properties
adding, 481-482
KeepCurrentDrawing, 483
NumberSquiggles, 483
property page, 483-485
SquiggleLength, 482
testing, 491-492
SquiggleLength property (Squiggle control), 482
SQL_FETCH_ABSOLUTE navigation type (Move function), 320
SQL_FETCH_FIRST navigation type (Move function), 320
SQL_FETCH_LAST navigation type (Move function), 320
SQL_FETCH_NEXT navigation type (Move function), 320

SQL_FETCH_PRIOR navigation type (Move function), 320
SQL_FETCH_RELATIVE navigation type (Move function), 320
stack
size, 433
viewing, 693-694
Standard toolbar, 10
start/end pages (print jobs), 647-650
StartDoc function, 659
starting applications
counting timer, 76-77
Menu Designer, 109
OnIdle tasks, 456
Run Program buttons, 42-43
threads, 462-463
StartPage function, 660
statements
catch, 663-664
enum, 571
if, 549
new, 666
return, 546
struct, 559
throw, 666
try, 663-664
typedef, 548
see also directives
static library project
class definitions, 385
drawing area, setting, 386
drawings
clearing, 390-391
creating, 386-389
displaying, 389-390
serializing, 390
project module, 384-385
test application, 392-395
static text control, 26
status bars
messages, 273-275
panes, 271-272
stepping through code, *see* single stepping
Stop buttons, creating
browser application, 532-533
counting timer, 79-81
Stop function, 524, 532
stopping
counting timer
OnStoptimer function, 77
Stop button, 79-81
OnIdle tasks, 456
print jobs, 657
threads, 462-463
strcpy function, 726
StretchBlt function, 155
strikeout effect (fonts), 127
string manipulation class
case sensitivity, 727
constructor types, 724
formatting, 728-729
manipulation functions, 727
NULL-terminated types, 726
searches, 727-728
size, 725
text, adding, 725
strlen function, 726
struct statement, 559
structures
accessing, 561
defined, 559
ENUMLOGFONTEX, 124
example, 560
LOGFONT, 123
NEWTEXTMETRICEX, 124
OPENFILENAME, 102
SubtractRect function, 714
suspended mode (threads), 433
SuspendSpinner function, 462-463
SuspendThread function, 469
synchronizing threads, 434-435
critical sections, 436
events, 439-440
mutexes, 436-438
semaphores, 438-439

T

tab order (controls), specifying, 30-31
Tab Order command (Layout menu), 30
Tasking application, *see* Spinner application
TBSTYLE_ALTDRAG toolbar control style, 250
TBSTYLE_AUTOSIZE toolbar button style, 252
TBSTYLE_BUTTON toolbar button style, 252
TBSTYLE_CHECK toolbar button style, 252
TBSTYLE_CHECKGROUP toolbar button style, 252
TBSTYLE_CUSTOMERASE toolbar control style, 250
TBSTYLE_DROPDOWN toolbar button style, 252
TBSTYLE_FLAT toolbar control style, 250
TBSTYLE_GROUP toolbar button style, 252
TBSTYLE_LIST toolbar control style, 250
TBSTYLE_NOPREFIX toolbar button style, 253

TBSTYLE_REGISTER-DROP toolbar control style, 250
TBSTYLE_SEP toolbar button style, 253
TBSTYLE_TOOLTIPS toolbar control style, 250
TBSTYLE_TRANSPARENT toolbar control style, 250
TBSTYLE_WRAPABLE toolbar control style, 250
templates, 707-710
Test Container (ActiveX controls), 491-492
TestApp application
- application shell, 393
- CTestAppDoc class, 423-424
- document functions
  - DeleteContents function, 425
  - GetDrawing, 425
  - OnNewDocument, 424
  - Serialize, 425
- drawings
  - creating, 395
  - deleting, 396
  - saving, 396
  - viewing, 397
- #include directives, 393
- view functions, 425-426

testing ActiveX controls, 491-492
text
- buttons, 276
- fonts
  - CreateFont function, 125-129
  - listing available, 122-124, 131-133
  - overview, 122
  - sample application, 129-131
  - selecting, 136-138
  - setting, 135-136
- status bars, 273-275
- strings, adding, 725

TEXT mapping mode, 156
ThreadFunc function, 461, 470
threads
- arrays, 433
- clean-up procedures, 466-468
- idle processing threads, 431
  - building, 457-459
  - continuous tasks, 459
  - OnIdle function, 455
  - starting, 456
  - stopping, 456
- independent threads, 431, 460
  - creating, 432
  - main thread function, 461
  - priorities, 432-433
  - stack size, 433
  - starting, 462-463
  - stopping, 462-463
  - suspended mode, 433
  - triggering from view object, 463-464
- synchronizing, 434-435
  - critical sections, 436
  - events, 439-440
  - mutexes, 436-438
  - semaphores, 438-439
- viewing, 697

throw keyword, 666
ThrowErrno function, 671
throwing exceptions, 666-667
ThrowOsError function, 671
tilde (~), 564
time-handling classes, 718-719
- COleDateTime class, 719-722
  - formatting code, 720-721
  - operator overloads, 722
- COleDateTimeSpan class, 722-724

timers
- clock timer
  - OnInitDialog function, 71-72
  - OnTimer function, 73
- counting timer
  - control variables, 74-75
  - OnChangeInterval function, 75
  - OnInitDialog function, 76
  - OnTimer function, 77-79
  - starting, 76-77
  - Stop button, 79-81
  - stopping, 77
- defined, 68
- IDs, 70-71
- interval range, 81
- maximum number of, 81

TIME_CRITICAL thread priority, 432
toolbars, 244-245
- attaching to application frames, 247-249
- Build toolbar, 10
- buttons
  - adding, 245
  - deleting, 245
  - icons, 246
  - styles, 252-253
  - text, 276
- combo boxes
  - creating, 260-266
  - event handling, 267-268
  - placeholders, 262
  - populating, 264
  - positioning, 263
  - resource files, 257-260
  - sizing, 276

styles, 263-264
updating, 268-270
creating, 246
borders, 251
child window IDs, 251
control styles, 250
CreateEx function, 250
IDs, 247
toolbar styles, 250-251
docking, 253-254
hiding, 254
OnUpdateViewColorbar function, 255
OnViewColorbar function, 256
ShowControlBar function, 255
ShowWindow function, 255
loading, 250
moving, 10
Standard toolbar, 10
status bars
adding panes, 271-272
messages, 273-275
toggling on/off, 10
WizardBar, 10

tools
MFC Tracer, 698
OLE/COM Object Viewer, 697
Process Viewer, 697
regsvr32, 183
Spy++ tool
Messages view, 695-696
Processes view, 697
Threads view, 697
Windows view, 696

TOOLTIPS toolbar control style, 250
TopLeft function, 712
TPM_CENTERALIGN flag, 118
TPM_LEFTALIGN flag, 118
TPM_LEFTBUTTON flag, 118
TPM_RIGHTBUTTON flag, 118
TRACE macros, 685-687, 708
listing, 686-687
output, 687
tracking mouse coordinates, 53-55
TrackPopupMenu function, 117
TRANSPARENT toolbar control style, 250
TrimLeft function, 727
TrimRight function, 727
troubleshooting
classes, 708-709
hash keys, 710
menu colors, 240
mnemonic conflicts, 31
resource leaks, 151
see also debugging techniques
TrueType fonts, listing available, 140
try keyword, 663-664
try...catch blocks, 207
turning on/off
edit boxes, 39-41
maximize/minimize buttons, 21-22
toolbars, 10
TWIPS mapping mode, 156
typedef statement, 548
typefaces, *see* fonts
types, *see* data types

U

underlined fonts, 127
Uniform Resource Locators, *see* URLs
UnionRect function, 713
unions, 714
Unlock function, 439-440
unlocking semaphores, 439
unsigned int data type, 548
unsigned long int data type, 548
unsigned short int data type, 548
Update function, 320, 356
UpdateBoundData function, 370-371
UpdateColorbar function, 269-270
UpdateData function, 35, 456
UpdateWidthCB function, 269
UPDATE_COMMAND_UI function, 225-227
updating
combo boxes, 268-270
records, 356
URLs (Uniform Resource Locators)
displaying, 529-530
specifying, 526-528
utilities, *see* tools

V

variables
controls, assigning, 32-33
CPrintInfo function, 642
data types
checking, 691-694
sizes, 554-555
values, 547-548
declaring, 548
default access permissions, 292

global, 552-553
local, 552-553
naming conventions, 33, 554
reference variables, 557-558
Variables window (debugger), 692-694
VERIFY macro, 687-689
View menu, 107
viewing drawings, 396
GetDrawing function, 397
OnDraw function, 397, 426
viewpoints, 448
views, 8
virtual key codes, 65
virtual memory, 702
visibility of toolbars, 254
menu updates, 255
toggling, 255-256

W-Z

WaitForSingleObject function, 463
Watch window (debugger), 692-694
Web browsers, *see* browsers
WebBrowse application
application shell, 525
buttons
Back, 531
Forward, 531
Refresh, 533-535
Stop, 532-533
URLs
displaying, 529-530
specifying, 526-528
weight (fonts), specifying, 126-127
Width function, 713
Window menu, 107
windows, *see* menus; dialog windows
Windows messages, *see* messages
WinExec function, 44
Winsock (Windows Sockets API), 497-498
connections
closing, 503, 517-519
opening, 500-501, 511-514
error handling, 504
event handling, 503-504
messages
listening for, 501
receiving, 502-503, 516-517
sending, 502, 515-516
sockets, creating, 498-500, 509-511
see also Sock application
WizardBar, 10
wizards
Class Wizard
Member Variables tab, 32
Message Maps tab, 34
WizardBar, 10
WM_CONTEXTMENU message, 237
WM_KEYDOWN message, 57
WM_KEYUP message, 57
WM_LBUTTONDBLCLK message, 48
WM_LBUTTONDOWN message, 48
WM_LBUTTONUP message, 48
WM_MOUSEMOVE message, 48
WM_MOUSEWHEEL message, 48
WM_RBUTTONDBLCLK message, 48
WM_RBUTTONDOWN message, 48
WM_RBUTTONUP message, 48
WM_TIMER message, 68
workspace
creating, 11-12, 541-543
defined, 8
WRAPABLE toolbar control style, 250
WS_CHILD style
combo boxes, 263
toolbars, 251
WS_DISABLED combo box style, 263
WS_GROUP combo box style, 263
WS_HSCROLL combo box style, 263
WS_TABSTOP combo box style, 263
WS_VISIBLE style
combo boxes, 263
toolbars, 251
WS_VSCROLL combo box style, 263
WYSIWYG (what you see is what you get) printing, 641