

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

## [Introduction](#)

## [About the Authors](#)

## [Part I—Basic Programming with MFC](#)

### [Chapter 1—An Introduction to MFC](#)

#### [Windows versus MFC](#)

#### [The Microsoft Foundation Classes](#)

#### [The \*CObject\* Class](#)

#### [The Application Class](#)

#### [The Window Classes](#)

##### [Frame Windows](#)

##### [View Windows](#)

##### [MDI Windows](#)

##### [Dialog Boxes](#)

##### [Property Sheets](#)

#### [Other Important MFC Classes](#)

##### [The \*CMenu\* Class](#)

##### [The \*CDC\* and \*CGDIObject\* Classes](#)

##### [The Control classes](#)

##### [The \*CArchive\* and \*CFile\* Classes](#)

##### [The Database Classes](#)

#### [The General Support Classes](#)

### [Chapter 2—Using AppWizard to Create an MFC Program](#)

#### [Understanding Wizards](#)

#### [Creating Your First MFC Program](#)

#### [Running Your First MFC Application](#)

#### [Exploring AppWizard's Files and Classes](#)

### [Chapter 3—Documents and Views](#)

Understanding the Document Class

Understanding the View Class

Creating the Rectangles Application

Creating the Basic Rectangles Application

Modifying the Document Class

Modifying the View Class

Running the Rectangles Application

Exploring the Rectangles Application

Declaring Storage for Document Data

Initializing Document Data

Serializing Document Data

Displaying Document Data

Modifying Document Data

## Chapter 4—Constructing an MFC Program from Scratch

Writing the Minimum MFC Program

Creating the Application Class

Creating the Frame-Window Class

Compiling an MFC Program

Responding to Windows Messages

Declaring a Message Map

Defining a Message Map

Writing Message Map Functions

Exploring the MFC App2 Application

## Chapter 5—Painting a Window

Understanding Device Contexts

Introducing the Paint1 Application

Exploring the Paint1 Application

Painting in an MFC Program

Using Fonts

Using Pens

Using Brushes

Switching the Display

Sizing and Positioning the Window

## Chapter 6—Using Menus

Understanding Menus

Creating a Menu Resource

Defining Menu IDs

Dealing with Resource Files

Introducing the Menu Application

Exploring the Menu Application

Declaring Menu State Variables

Declaring Menu Message Handlers

Defining the Message Map

Writing Menu Message Handlers

Writing Update-Command-UI Functions

## Chapter 7—Programming Dialog Boxes

Understanding Dialog Boxes

Creating a Dialog Box Resource

Defining Dialog Box and Control IDs

Looking at the Resource Script of a Dialog Box

Introducing the Dialog Application

Exploring the Dialog Application

Displaying a Simple Dialog Box

Writing a Dialog Box Class

Using the Dialog Box Class

## Chapter 8—Programming Controls

Understanding Controls

Introducing the Control1 Application

Exploring the Control1 Application

Declaring a Friend Function

Associating MFC Classes with Controls

Initializing the Dialog Box's Controls

Responding to the OK Button

Handling the Dialog Box in the Window Class

## Part II—Programming Windows 95 Controls

### Chapter 9—Progress Bar, Slider, and Spinner Controls

## The Win95 Controls Application

### The Progress Bar Control

#### Creating the Progress Bar

#### Initializing the Progress Bar

#### Manipulating the Progress Bar

### The Slider Control

#### Creating the Slider

#### Initializing the Slider

#### Manipulating the Slider

### The Spinner Control

#### Creating the Spinner Control

#### Initializing the Spinner Control

## Chapter10—Image List, List View, and Tree View Controls

### The Image List Control

#### Creating the Image List

#### Initializing the Image List

### The List View Control

#### Creating the List View

#### Initializing the List View

#### Associating the List View with Its Image Lists

#### Creating the List View's Columns

#### Creating the List View's Items

#### Manipulating the List View

### The Tree View Control

#### Creating the Tree View

#### Initializing the Tree View

#### Creating the Tree View's Items

#### Manipulating the Tree View

## Chapter11—Toolbars and Status Bars

### Working with Toolbars

#### Introducing the Toolbar Application

#### Creating the Toolbar Resource

#### Creating Toolbar Message Response Functions

#### Creating and Displaying a Toolbar



[Setting the Toolbar's Styles](#)

[Enabling Toolbar Docking](#)

[Showing and Hiding the Toolbar](#)

[Creating Toolbar Radio Buttons](#)

[Creating Custom Toolbars](#)

[Exploring the Custom Toolbar Class](#)

[Working with Status Bars](#)

[Understanding Status Bar Basics](#)

[Creating Custom Panes](#)

[Modifying a Panel's Appearance](#)

[Creating the Final Version of the Toolbar Application](#)

## **Chapter 12—Property Sheets and Wizards**

[Introducing Property Sheets](#)

[Creating the Property Sheet Demo Application](#)

[Creating the Basic Property Sheet Demo Application](#)

[Creating Resources for Property Pages](#)

[Creating Classes for Property Pages and Property Sheets](#)

[Completing the Application](#)

[Running the Property Sheet Demo Application](#)

[Understanding the Property Sheet Demo Application](#)

[Changing Property Sheets to Wizards](#)

[Running the Wizard Demo Application](#)

[Creating Wizard Pages](#)

[Setting the Wizard's Buttons](#)

[Responding to the Wizard's Buttons](#)

[Displaying a Wizard](#)

## **Chapter 13—The Rich Edit Control**

[Introducing the Rich Edit Control](#)

[Creating the RichEdit Application](#)

[Examining the RichEdit Application](#)

[Creating the Rich Edit Control](#)

[Initializing and Manipulating the Rich Edit Control](#)

## Chapter 14—The Animation Control

Introducing the Animation Control

Creating the Animate Application

Exploring the Animate Application

Creating the Animation Control

Manipulating the Animation Control

Adding an Animation Control to a Dialog Box

Exploring the Animate2 Application

## Part III—Advanced Programming with MFC

### Chapter 15—MFC File Handling

Objects and Persistence

The File Demo Application

A Review of Document Classes

A Quick Look at File Demo's Source Code

Creating a Persistence Class

The File Demo 2 Application

Looking at the *CMessages* Class

Using the *CMessages* Class in the Program

Reading and Writing Files Directly

The File Demo 3 Application

The *CFile* Class

Exploring the File Demo 3 Application

Creating Your Own *CArchive* Objects

### Chapter 16—Using Bitmaps

Introducing Device-Dependent Bitmaps

Creating the Bitmap Application

Exploring the Bitmap Application

Examining the *OnCreate()* Function

Examining the *OnDraw()* Function

Examining the *OnLButtonDown()* Function

Examining the *DrawSquare()* Function

## Chapter 17—Manipulating Device-Independent Bitmaps

### Reviewing DDBs and DIBs

#### The DIB Format

##### The *BITMAPFILEHEADER* Structure

##### The *BITMAPINFO* Structure

##### The *BITMAPINFOHEADER* Structure

##### The *RGBQUAD* Structure

### Introducing the CDib Class

#### The *CDib* Class's Interface

#### Programming the *CDib* Class

#### Loading a DIB into Memory

#### Other *CDib* Member Functions

### Creating ShowDib, Version 1

#### Creating the Basic Application

#### Modifying ShowDib's Resources

#### Adding Code to ShowDib, Version 1

#### Examining the *OnFileOpen()* Function

#### Examining the *OnDraw()* Function

#### Running Version 1 of ShowDib

### Creating ShowDib, Version 2

#### Mapping Between a Logical and a System Palette

#### Adding Code to ShowDib, Version 2

#### Examining the *CreateDibPalette()* Function

#### Examining Changes to the *OnDraw()* Function

#### Running Version 2 of ShowDib

### Creating ShowDib, Version 3

#### Responding to Palette Messages

#### Adding Code to ShowDib, Version 3

#### Examining the *OnPaletteChanged()* Function

#### Examining the *OnQueryNewPalette()* Function

## Chapter 18—Using the MFC Collection Classes

### The Array Classes

#### Introducing the Array Demo Application

#### Declaring and Initializing the Array

#### Adding Elements to the Array

Reading Through the Array

Removing Elements from the Array

### The List Classes

Introducing the List Demo Application

Declaring and Initializing the List

Adding a Node to the List

Deleting a Node from the List

Iterating Over the List

Cleaning Up the List

### The Map Classes

Introducing the Map Demo Application

Creating and Initializing the Map

Retrieving a Value from the Map

Iterating Over the Map

### Collection Class Templates

## Chapter 19—MFC Utility Classes

The *CString* Class

The *CTime* and *CTimeSpan* Classes

Using a *CTime* Object

Using a *CTimeSpan* Object

The *CPoint* Class

The *CSize* Class

The *CRect* Class

## Chapter 20—Programming Threads

Understanding Simple Threads

Understanding Thread Communication

Communicating with Global Variables

Communicating with User-Defined Messages

Communicating with Event Objects

Using Thread Synchronization

Using Critical Sections

Using Mutexes

Using Semaphores

## Chapter 21—Exceptions and Other Power Features

## Compiling and Running Console Applications

### Understanding Exceptions

#### Simple Exception Handling

#### Exception Objects

#### Placing the *catch* Block

#### Handling Multiple Types of Exceptions

### Exploring Templates

#### Introducing Templates

#### Creating Function Templates

#### Creating Class Templates

### Using Run-Time Type Information

#### Introducing RTTI

#### Performing Safe Downcasts

#### Getting Object Information

#### Preparing to Use RTTI

#### A Common Use for RTTI

### Namespaces

#### Defining a Namespace

#### Namespace Scope Resolution

#### Unnamed Namespaces

#### Namespace Aliases

## Chapter 22—Database Programming

### Understanding Database Concepts

#### Using the Flat Database Model

#### Using the Relational Database Model

#### Accessing a Database

#### Discovering the MFC ODBC Classes

### Creating an ODBC Database Program

#### Registering the Database

#### Creating the Basic Employee Application

#### Creating the Database Display

#### Adding and Deleting Records

#### Sorting and Filtering

### ODBC versus DAO

## Chapter 23—Global Variables and Macros

Ten Categories of Macros and Globals

Application Information and Management Functions

ClassWizard Comment Delimiters

Collection Class Helpers

CString Formatting and Message Box Display

Data Types

Diagnostic Services

Exception Processing

Message Map Macros

Runtime Object Model Services

Standard Command and Window IDs

## Chapter 24—Programming Internet Applications with WinInet

Introducing MFC's WinInet Classes

Programming a Web Application

Exploring the HTTPApp Application

Creating a CInternetSession Object

Creating a CHttpFile Object

Reading the HTML Document

Processing the HTML Document

Closing the File and the Internet Session

Programming an FTP Application

Exploring the FTPApp Application

Creating a CFtpConnection Object

Reading the FTP Server's Root Directory

Browsing the FTP Server

Downloading Files from an FTP Server

## Part IV—MFC and ActiveX

### Chapter 25—An Introduction to ActiveX

OLE

OLE 2

OLE and COM

OLE and ActiveX

ActiveX Today

## Chapter 26—Creating an ActiveX Container Application

Creating the Basic ActiveX Container Application

Embedding an Object

Linking an Object

Understanding the ActiveXCont1 Skeleton Application

Exploring the *CActiveXCont1App* Class

Exploring the *CActiveXCont1CntrItem* Class

Exploring the *CActiveXCont1View* Class

Exploring the *CActiveXCont1Doc* Class

Enabling Mouse Selection of Objects

Selecting ActiveX Objects

Understanding Part 2 of ActiveXCont1

Exploring Changes to the *ActiveXCont1CntrItem* Class

Exploring the *OnLButtonDown()* Function

Exploring the *OnLButtonDblClk()* Function

Exploring the *OnSetCursor()* Function

Exploring the *OnDraw()* Function

Exploring the *InitTracker()* Function

Exploring the *GetHitItem()* Function

Exploring the *SetSelectedItem()* Function

Deleting ActiveX Objects

Using the Third Version of ActiveXCont1

Exploring the *OnEditClear()* Function

## Chapter 27—Creating an ActiveX Server Application

Creating the Basic ActiveX Server Application

Understanding the ActiveXServ Skeleton Application

Exploring the *CActiveXServApp* Class

Exploring the *CActiveXServSrvrItem* Class

Exploring the *CActiveXServView* Class

Exploring the *CActiveXServDoc* Class

Exploring the *CInPlaceFrame* Class

Completing the ActiveXServ Server Application

Using the Final Version of ActiveXServ

## **Chapter 28—Discovering OLE Automation**

Creating an Automation Server Application

Creating the Basic AutoServer Application

Running the AutoServer Stand-alone Application

Exploring the *CAutoServerApp* Class

Exploring the *CAutoServerDoc* Class

Adding Properties and Methods to the  
Automation Server

Creating an Automation Client Application

Running the AutoClient Application

## **Chapter 29—Creating ActiveX Controls**

Introducing ActiveX Controls

Creating ActiveX Controls with MFC

Creating the Basic Calculator ActiveX Control

Running the Basic Calculator Control

Creating the Control's User Interface

Adding Properties to the Control

Testing the Calculator Control's Properties

Understanding Persistent Properties

Property Notification Functions

Adding a Button Notification Function

Testing the Button

Adding Methods to a Control

Testing the Method

Adding an ActiveX Control to a Web Page

## **Part V—Active Template Library (ATL)**

### **Chapter 30—Using the Active Template Library**

Introducing the ActiveX Template Library

How ATL Is Implemented

What ATL Is



What ATL Isn't

When to Use ATL

ATL versus MFC

Creating Your First ATL Project

Using the ATL COM AppWizard

Using the ATL Object Wizard

Generating Code with the Object Wizard

## Chapter 31—Creating ActiveX Controls with ATL

Creating a Basic Control

Registering ATL Controls

Testing Your Control with *TSTCON32*

Creating Methods

Adding Methods to the Control

The *ShowCaption* Implementation

Using Properties

Creating Normal User-Defined Properties

Creating Parameterized User-Defined Properties

Using Stock Properties

Using Ambient Properties

Creating Property Sheets

Responding to OLE Events

Adding the Event Interface

Defining an Event Method

Supporting Persistence

Drawing the Control

## Chapter 32—Advanced Programming with ATL

Adding Asynchronous Properties

Adding the *ReadyState* Property

Supporting the *ReadyStateChange* Event

Static and Dynamic Property Enumeration

Supporting Dynamic Property Enumeration

Getting the Right Property Page

Getting the Enumerations' String Values

- Getting an Enumeration Value
- Turning an Enumeration into a String
- Optimizing Control Drawing
- Adding Clipboard and Drag-and-Drop Support
  - Talking to the Clipboard
  - Supporting Drag-and-Drop
  - Handling Custom Clipboard and Drag-and-Drop Formats
- ATL Support for Other Advanced ActiveX Features
  - Dual-Interface Controls
  - Windowless Activation
  - Unclipped Device Context
  - Flicker-Free Activation
  - Mouse Pointer Notifications when Inactive
  - Optimized Drawing Code
- Summing Up

## Part VI—MFC and Computer Science

### Chapter 33—Programming Linked Lists

- The Story of Life
- The Rules of Life
- Life Implementation
  - Creating and Killing Cells
  - Dealing with the Speed Problem
- Linked Lists
  - Creating a Linked List
  - An Object-Oriented List
- The Life Program
- Examining Life
  - The Application and Dialog Box Classes
  - The Main Window Class
  - Creating the Window and Initializing Variables
  - The Toolbar and Status Bar
  - The Window's Bitmap
  - The Simulation's Main Loop

## **Chapter 34—Understanding Recursion**

**Recursion: Barrels Within Barrels**

**A Real-World Example of Recursion**

**A Power Function Using Recursion**

**Recursion and the Stack**

**An Example Application: Trap Hunt**

**Programming with Trees**

**Trap Hunt's Trees**

## **Chapter 35—Developing a Class**

**A Review of Object-Oriented Programming Techniques**

**Encapsulation**

**Classes as Data Types**

**Header Files and Implementation Files**

**Inheritance**

**Polymorphism**

**Classes: From General to Specific**

**Single-Instance Classes**

**Responsible Overloading**

**Overloading versus Default Arguments**

**Using Operator Overloading Logically**

**When to Use Virtual Functions**

**Developing a String Class**

**Choosing a Storage Method**

**Determining the Class's Functionality**

**String Construction and Destruction**

**String Assignments**

**String Concatenation**

**String Comparison**

**String Searches**

**String Insertion**

**String Deletion**

**String Extraction**

**String Retrieval**

**Testing the *String* Class**

**Conclusion**

# [Index](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Introduction

Programming under Windows is a lot like trying to find your way through a complex maze stocked with deadly traps and ravenous beasts. One wrong step and you're dead. The Windows maze is a dark place, indeed, so dark that it's darn near impossible to see where you're going. And every time you think you see the exit, something drags you back into the dark.

Because you bought this book, you're probably a veteran of the Windows dungeon. You've already dulled your programming blade battling thousands of pages of Windows documentation. You've treaded carefully through dank corridors paved with obscure technical jargon, stepped around secret trap doors (and probably fallen into a few) hiding incomprehensible program bugs, and yearned for just a glimmer of light—just a spark—to stave off the ever-encroaching darkness.

Luckily, people like you and I—programmers who want to get their work done more quickly and easily—can now see a light around the next bend. Microsoft has released version 5.0 of their popular Visual C++ development environment, and, thanks to Microsoft Foundation Classes (MFC), which is included with Visual C++, programming Windows applications with C++ has never been easier.

What is MFC? MFC is an object-oriented library that encapsulates almost the entire Windows API. Using MFC, you can display a fully functional window on-screen with only a few program lines. Moreover, Visual C++ includes MFC tools like AppWizard, ClassWizard, and ActiveX ControlWizard that make starting a new MFC project as easy as a few mouse clicks. This book also introduces you to the Active Template Library (ATL), a complement to MFC, which reduces the burden of programming Windows.

## Who This Book Is For

This book is not an introductory text for programmers interested in learning C++ programming. To understand the lessons included here, you must have a working knowledge of C++ and be somewhat familiar with Visual C++'s development system. In addition, you should have a good grasp of object-oriented programming concepts. Although previous Windows programming experience is helpful, you can probably get by without it. Still, you'll want to have a good Windows programming manual at your side as you work through the programs in this book.

# Hardware and Software Requirements

To compile and run the programs on this book's disk, and to get the most out of the upcoming lessons, you must have the following:

- An IBM-compatible 80486 or better with 16M of memory.
- Hard drive
- CD-ROM drive
- A Microsoft-compatible mouse
- Microsoft Windows 95 or Windows NT
- Visual C++ 5.0

As always, the faster your processor, the better. Fast processors mean fast compiles and zippy programs. This is especially true for Windows programs, because Windows pushes your hardware to the limits.

## How the Book Is Organized

This book is divided into parts, each of which concentrates on a specific area of MFC and ATL programming. The five parts feature many hands-on programming projects and are listed with a brief description of their contents:

- *Part I Basic Programming with MFC* This section, which includes Chapters 1 through 8, provides a comprehensive overview of using MFC to program Windows applications. Topics include using AppWizard, learning about MFC's Document/View architecture, and programming MFC applications without AppWizard, as well as programming windows, menus, dialog boxes, and controls.
- *Part II Programming Windows 95 Controls* This section, which includes Chapters 9 through 14, describes how to incorporate the new Windows 95 controls—including the slider, progress bar, list view, tree view, toolbar, status bar, rich edit, and animation controls—into your applications. Creating property sheets and wizards is also covered.
- *Part III Advanced Programming with MFC* This section, which includes Chapters 15 through 24, presents many advanced MFC programming techniques, including handling files, loading and displaying bitmaps, using MFC's collection and utility classes, programming threads, and writing Internet applications with WinInet. Also included are a chapter on database programming and chapters covering such advanced features as exceptions, templates, namespaces, RTTI, and MFC's many global variables, functions, and macros.
- *Part IV MFC and ActiveX* This section, which includes Chapters 25 through 29, introduces you to the MFC ActiveX classes. Covered here are server and client applications, as well as automation and ActiveX controls. Along the way, you'll learn about data linking and embedding, as well as how to use ActiveX controls in your own Web pages.
- *Part V Active Template Library (ATL)* This section, which includes Chapters 30 through 32, introduces you to ATL, comparing and contrasting it with MFC to show how the two complement one another.

You will learn how to add methods, properties, and events, as well as how to support property sheets and persistent properties, as you build an ActiveX control by using only ATL. Finally, Chapter 32 explores advanced programming with ATL, teaching you how to implement optimized drawing and asynchronous properties.

- *Part VI MFC and Computer Science* This section, which includes Chapters 33 through 35, presents valuable programming techniques that you can incorporate into your MFC programs. Topics include linked lists, recursion, and designing your own classes. Two complete games—Conway’s Game of Life and Trap Hunter—demonstrate the described programming techniques.

## Compiling the Programs in this Book

The programs in this book were written with Visual C++ 5.0. This book assumes that your copy of Visual C++ was installed using the default settings and directories. If you’ve changed any of the default settings or directories and are not sure how to fix errors that may result from these changes, you should reinstall Visual C++.

The programs that follow are organized on the book’s CD-ROM by chapter. Each chapter’s programs are found in their own directory on the CD-ROM. The programs in Chapter 2 are in the CHAP02 directory; the programs in Chapter 3 are in the CHAP03 directory, and so on. To compile the programs for a specific chapter, copy all of the files from the chapter’s directory to your hard drive. Then start Visual C++ and load the .dsw workspace file for the program that you want to compile. Finally, choose the Build, Build command to compile and link the program.

For further information on creating and building projects, please refer to your Visual C++ 5.0 documentation.

## A Word to the Wise

As every developer knows, a good program is virtually crash proof. Error-checking must be done for every action that might fail, and appropriate error messages must be given to the user. Unfortunately, good error-checking requires a lot of extra program code. For the programmer working on his next magnum opus, this is all just part of the game. But for an author writing a programming book, this extra code has different implications.

A programming book should present its topics in as clear a manner as possible. This means featuring programs with source code not obscured by a lot of details that don’t apply directly to the topic at hand. For this reason, the programs in this book do not always employ proper error-checking. User input might sometimes go unverified, dynamic construction of objects is assumed to be successful, and (horror of horrors) pointers are not always checked for validity.

In short, if you use any of the code in this book in your own programs, it’s up to you to add whatever error-checking might have been left out. Never assume

anything in your programs. In any place in your code that you aren't 100 percent confident of your program's state, you must add error-checking to ensure that the program doesn't come crashing down on your users. Just because this book's author may have been lax in his error-checking (for good reasons), does not let you off the hook.

## Get Started with MFC and ATL

Now that you have some idea of what's ahead, it's time to start programming with MFC and ATL. You'll soon discover that, not only do MFC and ATL programming take much of the sting out of creating Windows applications, but they also enhance your understanding of object-oriented programming techniques. And, best of all, programming with MFC and ATL is fun.

## Conventions Used in This Book

Que has over a decade of experience developing and publishing the most successful computer books available. With that experience, we've learned what special features help readers the most. Look for these special features throughout the book to enhance your learning experience.

Several type and font conventions are used in this book to help making reading it easier:

- *Italic type* is used to emphasize the author's points and to introduce new terms.
- Screen messages, code listings, and command samples appear in monospace typeface.
- URLs, newsgroups, Internet addresses, and anything you are asked to type appears in **boldface**.

---

### Tip:

Tips present short advice on a quick or often overlooked procedure. These include shortcuts that can save you time.

---

---

### Note:

Notes provide additional information that may help you avoid problems, or offer advice that relates to the topic.

---

---

### Caution:

Cautions warn you about potential problems that a procedure may cause, unexpected results, and mistakes to avoid.

---

- **See** these cross-references for more information on a particular topic.



[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## About the Authors

**Clayton Walnum**, who has a degree in computer science, has been writing about computers for almost 15 years and has published hundreds of articles in major computer publications. He is also the author of over 25 books, which cover such diverse topics as programming, computer gaming, and application programs. His most recent book is *Using Java Workshop*, also published by Que. His other titles include the award-winning *Building Windows 95 Applications with Visual Basic* (Que), *Windows 95 Game SDK Strategy Guide* (Que), *Dungeons of Discovery* (Que), *PC Picasso: A Child's Computer Drawing Kit* (Sams), *Powermonger: The Official Strategy Guide* (Prima), *DataMania: A Child's Computer Organizer* (Alpha Kids), and *Adventures in Artificial Life* (Que). Mr. Walnum lives in Connecticut with his wife Lynn and their four children, Christopher, Justin, Stephen, and Caitlynn. His home page is located at <http://www.connix.com/~cwalnum>.

**Paul Robichaux**, who has been an Internet user since 1986 and a software developer since 1983, is currently a software developer for LJI Enterprises (<<http://www.ljl.com>>), where he writes data security and cryptography software for Win32 and MacOS platforms. In his spare time, he writes computer books and develops Macintosh applications; he still manages to spend plenty of time with his wife and young son. He can be reached via e-mail at [paulr@hiwaay.net](mailto:paulr@hiwaay.net).

## Acknowledgments

Sincere thanks go out to all the fine folks at Que for their help and support over the course of this project. Of special note are Jeff Riley, Juliet MacLean, Daniel Howard, Fred Slone, and Joe Wikert. And, as always, thanks to my family—Lynn, Christopher, Justin, Stephen, and Caitlynn.

## We'd Like to Hear from You!

As part of our continuing effort to produce books of the highest possible quality, Que would like to hear your comments. To stay competitive, we *really* want you to let us know what you like or dislike most about this book or other Que products.

Please send your comments, ideas, and suggestions for improvement to:

The Expert User Team

Brief Full

- Advanced
- Search
- Search Tips

 **BROWSE**  
BY TOPIC

E-mail: [euteam@que.mcp.com](mailto:euteam@que.mcp.com)

CompuServe: 105527,745

Fax: (317) 581-4663

Our mailing address is:

Expert User Team

Que Corporation

201 W. 103rd Street

Indianapolis, Indiana 46290-1097

You can also visit our Team's home page on the World Wide Web at:

[http://www.mcp.com/que/developer\\_expert](http://www.mcp.com/que/developer_expert)

Thank you in advance. Your comments will help us to continue publishing the best books available in today's market.

Thank You,

The Expert User Team

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## *Part I*

# *Basic Programming with MFC*

## Chapter 1

# An Introduction to MFC

- Understand how MFC helps you create Windows applications
- Learn how CWinApp, manages the tasks required by Windows applications
- Discover MFC's many window classes
- Learn about other useful MFC classes, graphical objects, and more

Remember when you bought your first VCR? If you got your VCR back when I got mine, it wasn't much more difficult to operate than a standard audio cassette recorder. The VCR had all the standard controls like play, fast forward, rewind, and record. If you were really lucky, it even had freeze-frame.

If you've purchased a VCR recently, however, you know that there are so many controls and settings that you need a degree in engineering to figure out everything. Today's VCRs include not only freeze-frame, but also slow motion, indexing, multiple-event timers, simulcast recording, multichannel sound, auto repeat, and on-screen clocks. I don't know about you, but to make sure that my recordings come out right, I've resorted to using a checklist to verify that each setting on the VCR is correct.

Programming a computer is a lot like programming a VCR. A few years ago, you could sit down in front of your computer with a simple language like BASIC and a slim volume of instructions and write an impressive program (at least it was impressive for those days). Today, however, if you want to write commercial quality software, you must use sophisticated, object-oriented languages such as C++, and you need enough documentation to cover your desk a foot deep.

Such is progress.

Luckily for today's programmers, while the languages and operating systems are becoming more and more complex, the programming tools are becoming more and more sophisticated. The major Windows compilers today come with

code libraries that attempt to make today's programming tasks quicker and easier. This brings us, of course, to the Microsoft Foundation Classes (MFC).

MFC is a C++ code library that simplifies the writing of Windows applications. If you've ever tried to write a Windows program without the help of MFC, you know what an immense task this can be. Using standard C++ programming, you'll need 80 or 90 lines of code just to get an empty window on-screen. A full-fledged Windows application can be immense beyond belief. Using MFC, however, you can get your first window on-screen with only a few lines of code.

## **Windows versus MFC**

Although Windows is complex, it is a veritable workhorse that handles much of your application's activity automatically. Windows applications usually have a main window with a menu bar, scroll bars, sizing buttons, and other controls—all of which are handled to a great extent by Windows. For example, when you create a window with a menu bar, your program doesn't need to control the menu. Windows does this for you, sending your program a message whenever the user selects an item in the menu.

A Windows program can receive hundreds of different messages while it's running. Your application determines whether to respond to these messages or to ignore them. If the user clicks the mouse pointer in your window, for example, your program gets a message (WM\_LBUTTONDOWN). If the program determines that the user clicked something important, the program can handle the message, performing the function the user requested. On the other hand, the program can simply ignore the message and let Windows take care of it. It's up to you.

All of this sounds terrific until you get your first look at a Windows programming manual and see the huge number of functions included in the Application Programming Interface (API). Surely there must be an easier way to program Windows than to plow through thousands of pages of documentation, isn't there?

Yes and no. No matter what route you take, learning to program Windows, although not especially difficult, takes a lot of time and practice. Before you program your first application, you should be familiar with at least the most used functions in the API so that you know the tools you have at your disposal.

However, Microsoft's MFC goes a long way toward simplifying the process of writing Windows applications by hiding many of the details inside custom window classes. As stated earlier in this chapter, by using MFC, you can create a fully operational window with very few lines of code. Of course, learning to use MFC is no picnic, either. MFC has its own set of rules and requirements in addition to those of Windows. This book teaches you those rules and requirements.

## **The Microsoft Foundation Classes**

MFC includes many classes you can use to write your applications more

quickly and easily. These classes represent objects from which a Windows application is created—objects such as windows, dialog boxes, menu bars, window controls, and many more, including some special objects such as status bars and control bars. In addition, MFC provides many general purpose classes for handling things like strings, arrays, and linked lists.

Specifically, MFC provides the following main categories of classes:

- Applications
- Windows
- Menus
- Dialog boxes
- Documents and views
- Controls
- Graphics
- Archives and file
- Database
- Various support classes

In the following sections, you get a brief look at each of these MFC class categories. If some of the descriptions are confusing at first, don't become discouraged. Most of the classes described in the following sections are covered in detail later in the book, at which point everything will become clear.

## The *CObject* Class

If you examine the MFC class hierarchy, you'll see that almost every class in the library is derived from the *CObject* class. You might think that a class that is the granddaddy to almost the entire MFC class library would be huge and complex. Ironically, though, *CObject* is one of the smallest classes in MFC. Of course, if you think a minute, this makes sense. *CObject* is the base for most MFC classes, so it must contain only the functionality common to all of the classes. With so many diverse classes in MFC, this commonality must be very general in nature.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 1.1 Member Functions of the CObject Class

Function	Description
AssertValid()	Checks whether an object is in a valid state
CObject()	Constructs a CObject object
~CObject()	Destructs a CObject object
Dump()	Displays debugging information about the class
GetRuntimeClass()	Returns information such as the class’s name and size
IsKindOf()	Checks whether an object is derived from a given class
IsSerializable()	Checks whether the object can perform serialization
Serialize()	Performs the object’s serialization duties

NOTE:

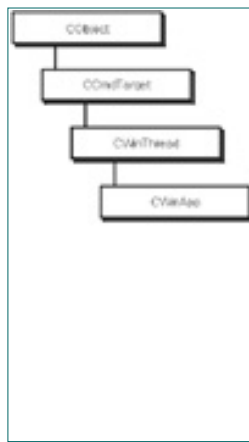
The word *serialize* is just a fancy term for the act of sending or retrieving data to or from some sort of storage. For example, you might have a class that needs to save and restore the data that the user entered into a form. To do this, you would override CObject’s Serialize() function in your class derived (directly or indirectly) from CObject. In Serialize(), you would place the code that actually saves or reads the object’s data. As you’ll see in Chapter 3, “Documents and Views,” MFC supplies the CArchive class, which makes saving and reading data unbelievably easy.

Although CObject acts as the base class for the majority of MFC classes, you can derive your own custom classes from CObject and, thus, automatically acquire built-in support for the functionality built into CObject—most notably the capability to serialize your custom object. Any custom class you want to develop that needs to save and load data should be derived, directly or indirectly, from CObject.

The Application Class

Every MFC program begins life as an application object. You derive this application object from MFC’s CWinApp class, which provides initialization, runtime, and ending services for your Windows program. More specifically, CWinApp registers, creates, and shows an application’s main window; sets the application’s message loop running (so that the application can interact with the user and Windows); and deletes the application when the user is finished. CWinApp also provides some additional services, such as providing access to the application’s command line, handling file activity, processing Windows messages, and detecting when Windows is idle. Every MFC program instantiates a CWinApp-derived object.

In the MFC hierarchy, CWinApp is derived from CWinThread, which is, in turn, derived from CCmdTarget (see Figure 1.1). The CWinThread class takes care of thread-related tasks. Because every process running under Windows 95 requires at least one thread of execution, CWinApp is derived from CWinThread. However, CWinThread also provides the power needed to build multithreaded applications.



**FIG. 1.1** The CWinApp class is derived from CWinThread.

---

**NOTE:**

A *thread* is nothing more than a subprocess—or, more specifically, a path of execution within a process. Your application is a process that contains at least one thread, which is the main thread. Your application can also contain a number of other threads, each of which performs some useful task on behalf of the application. For example, you might want to create a thread that recalculates some values while the main thread continues to gather user input. In a way, you can think of threads as a way of implementing multitasking from within an application, rather than a system, context.

---

Finally, all MFC classes that must respond to Windows messages can trace their ancestry back to CCmdTarget, which handles MFC's message mapping. Message mapping is the process of matching member functions in a class to the Windows messages to which the functions must respond. For example, MFC can map the WM\_LBUTTONDOWN message to a message-response function called OnLButtonDown(), which enables your application to automatically respond to left mouse-button clicks without having to resort to clumsy switch statements like those used in a conventional C Windows program.

Thanks to its rich heritage, the CWinApp class can do a lot of work for your application. Its public data members contain handy information about an application, and its member functions enable your application to do everything from create a main window to handle file commands. Table 1.2 lists the most useful CWinApp's public data members, whereas Table 1.3 lists some of CWinApp's most useful member functions. Look over these tables in order to get an idea about this class works.



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

In fact, CObject includes only eight member functions, including its constructor and destructor. These member functions enable the class to perform basic serialization (saving and reading data from and to storage), to provide run-time class information, and to output diagnostic information. To give you a brief overview of the class, the most important CObject member functions and what they do are shown in Table 1.1. Six other functions (not listed in the table) are overloaded C++ operators. They are not directly useful to the average programmer; they are required by MFC to make MFC more efficient and to prevent programming errors. The remaining method, GetBaseClass(), is used internally by MFC.

[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 1.2 Public Data Members of the CWinApp Class**

Data Member	Description
m_bHelpMode	Indicates whether the application is in Help context mode
m_hInstance	The current instance's handle
m_hPrevInstance	The previous instance's handle
m_lpCmdLine	A pointer to the application's command line
m_nCmdShow	A value indicating how the window is initially displayed
m_pMainWindow	A pointer to the application's main frame window
m_pszAppName	A pointer to the application's name
m_pszExeName	A pointer to the application's module name
m_pszHelpFilePath	A pointer to a string holding the application's help-file path
m_pszProfileName	A pointer to the application's .INI filename

**Table 1.3 Handy CWinApp Member Functions**

Function	Description
ExitInstance()	Performs shutdown cleanup for an instance of the application
GetProfileString()	Loads a string from the application's .INI file
InitInstance()	Performs start-up initialization for an instance of the application
LoadCursor()	Loads a cursor for the application
LoadIcon()	Loads an icon for the application
OnIdle()	Performs idle-time processing
ParseCommandLine()	Parses command lines into individual parameters
ProcessShellCommand()	Handles command lines used to run an application
SaveAllModified()	Reminds the user to save any changed documents
SetRegistryKey()	Stores application settings in the Registry
WriteProfileString()	Saves a string to the application's .INI file

**Note:**

Many of the functions listed in Table 1.3 are virtual functions that you override in the class that you derived from CWinApp. For example, when you want to provide an instance of your application with start-up initialization, you would override `InitInstance()` in your application class. Similarly, to perform last minute cleanup for an instance of your application, you'd override the `ExitInstance()` function. By overriding virtual functions of an MFC class, you can customize the class to perform exactly as you want it to.

# The Window Classes

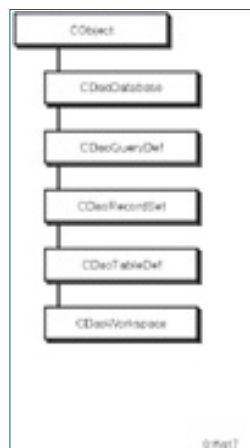
When you create a basic MFC application, it's up to you to tell MFC the type of main window it should create for the application. This is where MFC's window classes come in. By forcing your MFC application to create your custom main window, you can add all the functionality you need to your application. To make this task easier for you, MFC features several different types of windows. These windows include the following:

- Frame windows
- View windows
- MDI windows
- Dialog boxes
- Property sheets
- Wizards

All of the window classes are derived from the `CWnd` base class, which supplies the basic functionality required by any window class. This functionality includes initialization, positioning and sizing, painting, coordinate mapping, scrolling, message handling, and much more. There are, in fact, over 250 member functions in the `CWnd` class, far too many to list in this chapter. The `CWnd` class, however, has only one public data member, `m_hWnd`, which is the handle of the window with which the class is associated. Although you can derive your own window classes directly from `CWnd`, you're more likely to use one of MFC's specialized window classes. Some of these window classes are discussed in the sections that follow.

## Frame Windows

Frame windows, represented by the `CFrameWnd` class, supply all the functionality of the `CWnd` class (it is, after all, directly derived from `CWnd`, as shown in Figure 1.2), and also handle document/view matters, the menu bar, control bars, accelerators, and more. Frame windows also act as the base class from which four other MFC window classes are derived. These classes are `CMDIChildWnd`, `CMDIFrameWnd`, `CMiniFrameWnd`, and `COleIPFrameWnd`.



**FIG. 1.2** The `CFrameWnd` class is derived from `CWnd`.

The `CFrameWnd` class contains only two public data members, `m_bAutoMenuEnable` and `rectDefault`, which controls whether menu items are enabled or disabled automatically and controls the window's starting size and position, respectively. Table 1.4 lists some of the more important `CFrameWnd` class's member functions.



[Brief](#)   [Full](#)  
◆ [Advanced Search](#)  
◆ [Search Tips](#)



[an error occurred while processing this directive]

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

**Table 1.4 Member Functions of the CFrameWnd Class**

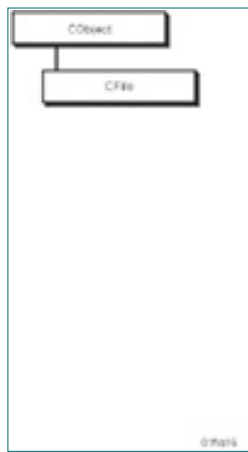
Function	Description
ActivateFrame()	Activates the frame window
CFrameWnd()	Constructs a CFrameWnd object
Create()	Creates the frame window
GetActiveDocument()	Returns a pointer to the active document
GetActiveFrame()	Returns a pointer to the active frame
GetActiveView()	Returns a pointer to the active view
LoadAccelTable()	Loads an accelerator table
LoadFrame()	Creates a frame window using resource information
OnContextHelp()	Enables context-sensitive help
OnCreateClient()	Creates a client window for the frame window
OnSetPreviewMode()	Controls print-preview mode
RecalcLayout()	Positions the frame window’s control bars
SetActiveView()	Activates a view object

In most cases, you’ll derive the main window class for an SDI (single-document interface) application from the CFrameWnd class. Such a frame window usually has a complete set of window controls—such as minimize, maximize, restore, and close buttons—and holds the application’s menu bar, toolbar, and status bar. However, MFC is flexible enough that you can create just about any type of frame window that you need from the CFrameWnd class.

## View Windows

To make it easier to handle the data associated with an application’s window, MFC offers the document/view architecture. Although “document/view architecture” sounds kind of scary, it’s really little more than a way to separate data from the way an application displays that data. The CDocument class (which boasts CCmdTarget and CObject as ancestors) is responsible for storing a window’s data, whereas the CView class (which traces its ancestry down through CWnd, CCmdTarget, and finally CObject) displays the data, as well as enables the user to manipulate the data.

The CView class (see Figure 1.3) provides all the functionality needed by a generic view window. This includes displaying data in the frame window’s client area and printing data. A document can be associated with more than one view. For example, you could have a document that stores plain text. This document could be associated with a view that displays the text normally and another view that displays the text as hexadecimal ASCII values. Although a document can have more than one view, an individual view must always be associated with only a single document. Table 1.5 lists the more useful CView class’s member functions and what they do.



**FIG. 1.3** The CView class is derived from CWnd.

**Table 1.5 Member Functions of the CView Class**

Function	Description
CView()	Constructs a CView object
DoPreparePrinting()	Initializes a print context
GetDocument()	Retrieves the document attached to the view
IsSelected()	Determines whether an OLE item in a document is selected
OnActivateView()	Called whenever the view is activated
OnBeginPrinting()	Called at the start of a print job
OnDraw()	Creates the view of the document, usually in a window
OnEndPrinting()	Called at the end of a print job
OnEndPrintPreview()	Called when the user leaves print preview
OnPrepareDC()	Called before OnDraw() or OnPrint() to modify the device context or handle pagination
OnPreparePrinting()	Initializes the Print dialog box
OnPrint()	Prints a page of the document or displays the print preview
OnUpdate()	Called when the document attached to the view has changed

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

[Brief](#)   [Full](#)

- ✚ [Advanced Search](#)
- ✚ [Search Tips](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

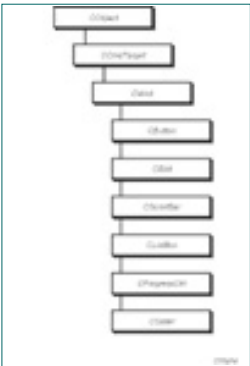
Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 1.6 Member Functions of the CMDIFrameWnd Class

Function	Description
CMDIFrameWnd()	Constructs a CMDIFrameWnd object
CreateClient()	Creates a client window for a frame window
GetWindowPopupMenu()	Returns a handle to the Window menu
MDIActivate()	Activates a child (document) window
MDICascade()	Places all child windows into a cascaded arrangement
MDIGetActive()	Returns a pointer to the active child window
MDIIconArrange()	Arranges document icons within the client window
MDIMaximize()	Maximizes a child window
MDINext()	Activates the next child window (going by Z order)
MDIRestore()	Restores a child window to its original size
MDISetMenu()	Changes the frame window's menu
MDITile()	Places all child windows into a tiled arrangement

In order to display the different documents associated with the MDI application, you need to create MDI child windows, which are represented in MFC by the CMDIChildWnd class. The CMDIChildWnd class is found on the same level of the MFC hierarchy as the CMDIFrameWnd class (Figure 1.5), tracing its ancestors from CFrameWnd, back through CWnd, CCmdTarget, and CObject. You can tell by this hierarchy that windows of the CMDIChildWnd class must be a lot like CFrameWnd frame windows. This is because MDI child windows have much the same relationship with their MDI frame window as SDI frame windows have with the Windows desktop. Just like regular frame windows, MDI child windows can contain view windows (instantiated from CView) that depict the data stored in a document object (instantiated from CDocument).



**FIG. 1.5** The CMDIChildWnd Class is Derived from CFrameWnd.

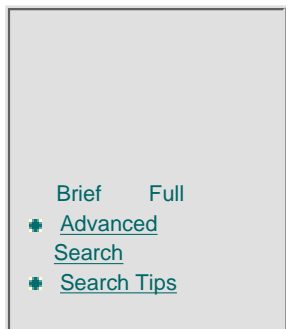
Thanks to object-oriented programming (OOP) inheritance, the CMDIChildWnd class gets most of its functionality from the CFrameWnd class. However, the CMDIChildWnd class adds the abilities to share a menu between MDI child windows associated with the same document (this is the menu that replaces the default frame-window menu whenever an MDI child window is active) and to handle changes to the frame window, depending on the active MDI child window. An example of such a change is placing the document's name in the frame window's title bar. Table 1.7 lists the member functions of the



**Table 1.7 Member Functions of the CMDIChildWnd Class**

Function	Description
CMDIChildWnd()	Constructs a CMDIChildWnd object
Create()	Creates an MDI child window
GetMDIFrame()	Returns a pointer to the MDI child window's frame window
MDIActivate()	Activates the MDI child window
MDIDestroy()	Destroys the MDI child window
MDIMaximize()	Maximizes the MDI child window
MDIRestore()	Restores the MDI child window to its original size

[Previous](#) [Table of Contents](#) [Next](#)



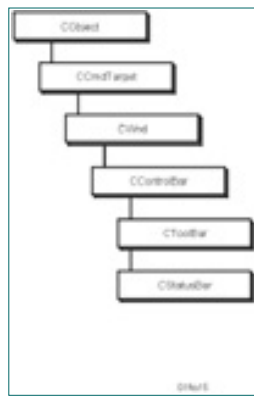
[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

As you may have guessed, MFC derives many specialized view classes from the general CView class. These include CCtrlView, CEditView, CListView, CRichEditView, CTreeView, CScrollView, CFormView, CDaoRecordView, and CRecordView. You can tell by the class names what many of these classes do. You'll also run into many of them later in this book. Of course, you can derive your own custom view classes from CView.

## MDI Windows

Most document-oriented Windows applications enable the user to open more than one document simultaneously. Such applications use Windows' MDI (multiple-document interface) to handle the extra chores that must be performed to handle several documents concurrently. MFC provides the CMDIFrameWnd class (which traces its lineage back through the CFrameWnd, CWnd, CCmdTarget, and CObject classes, as shown in Figure 1.4) to encapsulate the extra functionality needed by MDI frame windows. This extra functionality includes handling the MDI client window, which provides an area for MDI child (document) windows; managing the main and child-window menu bars; forwarding messages to MDI child windows; and arranging MDI child windows within the client window (that is, responding to Tile, Cascade, and Arrange commands found on an MDI application's Window menu). Table 1.6 lists the CMDIFrameWnd class's member functions.



**FIG. 1.4** The CMDIFrameWnd class is derived from CFrameWnd.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

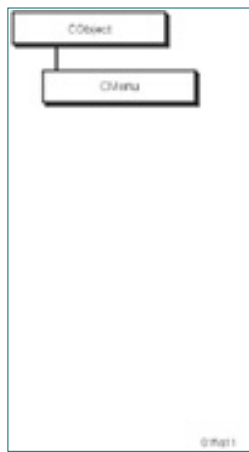
Table 1.8 Member Functions of the CDialog Class

Function	Description
CDialog()	Constructs a CDialog object
Create()	Creates a modeless dialog box associated with the class
CreateIndirect()	Creates a modeless dialog box from a template stored in memory
DoModal()	Displays a modal dialog box
EndDialog()	Closes a modal dialog box
GetDefID()	Returns the default button control's ID
GotoDlgCtrl()	Moves the focus to a given dialog box control
InitModalIndirect()	Creates a modal dialog box from a template stored in memory
MapDialogRect()	Converts dialog box measurements to screen measurements
NextDlgCtrl()	Moves the focus to the next dialog box control
OnCancel()	Responds to the dialog box's Cancel button
OnInitDialog()	Initializes a dialog box immediately before it's displayed
OnOK()	Responds to the dialog box's OK button
OnSetFont()	Sets a dialog box's font
PrevDlgCtrl()	Moves the focus to the previous dialog box control
SetDefID()	Sets the default button for the dialog box
SetHelpID()	Sets the dialog box's context-sensitive help ID

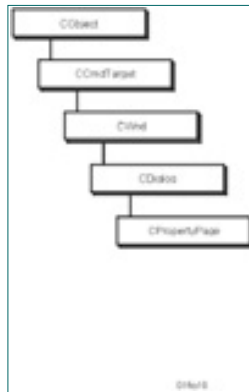
Property Sheets

Windows 95 uses a lot of special multipage dialog boxes called property sheets (see Figure 1.8). Because a property sheet features tab controls, the user can easily flip from page to page and quickly find the information needed, without having to decipher a cluttered dialog box or search through a series of standard one-page dialog boxes. MFC supports property sheets through its class CPropertySheet, which is derived from CWnd as shown in Figure 1.9.

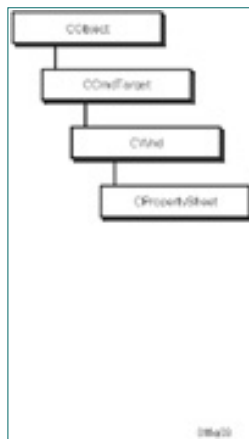
The CPropertySheet class works in conjunction with the CPropertyPage class, which represents each page in your property sheet. Because each page of a property sheet is very similar to a dialog box, you might not be surprised to learn that the CPropertyPage class is derived from CDialog, as shown in Figure 1.10. To create a property sheet, you use a dialog box editor to create each page. You then derive a class from CPropertyPage for each page of the property sheet, associating each derived class with a property sheet page. Finally, you create the CPropertySheet object that will hold the page objects.



**FIG. 1.8** Property sheets organize information into tabbed pages.



**FIG. 1.9** The CPropertySheet class is derived from CWnd.



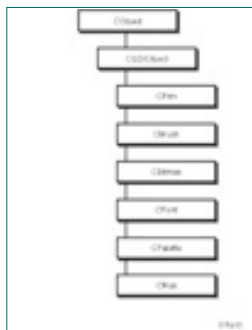
**FIG. 1.10** The CPropertyPage class is derived from CDialog.

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

## Dialog Boxes

Most Windows applications rely on dialog boxes to retrieve information from the user. Because dialog boxes are so ubiquitous in Windows programming, MFC features a whole set of classes dedicated to these specialized windows. MFC features a basic dialog box class, CDialog (see Figure 1.6), which handles all the details of constructing, executing, and closing a dialog box, including built-in responses for the most often used buttons and a mechanism for transferring data to and from the dialog box. In addition, your MFC dialog boxes can be modal or modeless and can contain any number of controls.



**FIG. 1.6** The CDialog class is derived from CWnd.

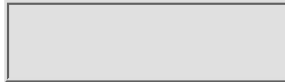
MFC also includes classes for Windows' common dialog boxes and other "prefab" dialog boxes. These dialog classes—including CColorDialog, CFileDialog, CFindReplaceDialog, CFontDialog, and CPrintDialog—enable users to select file names, find text strings, set up a printer, and choose colors and fonts. The common dialog classes are derived from CCommonDialog (see Figure 1.7), which is itself derived from CDialog.

You actually create your dialog box using a resource editor such as the dialog box editor included as part of the Microsoft Developer Studio, which comes with Visual C++. The dialog box you create with the editor comprises the visual elements of the dialog box, including the dialog box's main window and all of the controls that window contains. To add MFC functionality to the dialog box, you then derive a class from CDialog, attaching that class to the dialog box template you created with the dialog box editor.

To pass data to and from the dialog box, you create data members for the class and associate those data members with the controls for which they are to store data. You can then initialize the dialog box controls by setting these data members. Similarly, you can retrieve data from the dialog box's controls after it's closed by checking the contents of the associated data members. Table 1.8 lists CDialog's member functions.



[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Other Important MFC Classes

Although the various MFC window classes are the most used classes in the library, in order to create a full-fledged Windows application, those window classes rely heavily on the many classes that encapsulate Windows' menus, graphics, and controls. In addition, many applications will draw upon the MFC file and database classes. In the sections that follow, you get a quick look at these important classes.

### The *CMenu* Class

Almost every Windows application sports some sort of menu bar, so it only stands to reason that MFC would have a special class for dealing with menus. That class is *CMenu*, which enables an application to manipulate a Windows menu in various ways, including associating a menu with the class, creating and destroying menus, checking and enabling/disabling menu items, adding bitmaps to menu items, and more. As shown in Figure 1.11, the *CMenu* class is derived from *CObject*.



**FIG. 1.11** The *CMenu* class is derived from *CObject*.

### The *CDC* and *CGDIObject* Classes

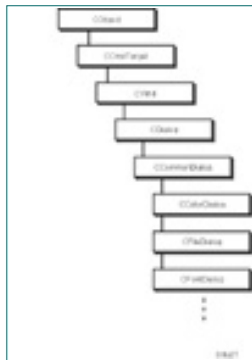
In order to be useful, most Windows applications have to display some sort of data in their windows. A word processor, for example, must display the text of the currently loaded document, whereas a graphics editor must display the bitmap on which the user is currently working. To display data in a window, an application must obtain a device context (DC) and then call functions of Windows' Graphics Device Interface (GDI) to display graphics on the surface represented by the DC. MFC's huge *CDC* class represents DCs, whereas the *CGDIObject* class (and all of the classes derived from *CGDIObject*) represent the objects such as pens and brushes that can be used with a DC.

The *CDC* class is derived directly from *CObject* and features hundreds of member functions that, not only encapsulate most of the GDI's functions, but also add some handy functionality of their own. Although you can create *CDC* objects in your programs, you'll most often create objects from one of the

classes derived from CDC (see Figure 1.12). These classes—CPaintDC, CClientDC, CWindowDC, and CMetaFileDC—provide special DCs for specific uses. For example, the CClientDC class represents a DC for the client area of a window, whereas CWindowDC provides access to the entire window, including both its client and nonclient areas.

**Note:**

A device context is nothing more than a collection of attributes that determines the type of surface that you can draw on and the specific objects—including pens, brushes, and fonts—you can use to do that drawing. Each of these drawing objects has its own attributes that are controlled by the DC. For example, when a DC is first created, it contains a default black pen for drawing lines. If you want to draw red lines, you must create a red pen and replace the black pen with the newly created red pen.



**FIG. 1.12** The CDC class is derived from CObject.

CGDIObj is the base class for MFC's drawing-object classes. As you can see from Figure 1.13, these drawing-object classes are, CPen, CBrush, CBitmap, CFont, CPalette, and CRgn and are derived from CGDIObj, which is in turn derived from CObject. To use MFC's classes to draw data in a window, you first create the appropriate CDC object, create and select into the DC the drawing objects you need, and then use the objects to do the drawing.

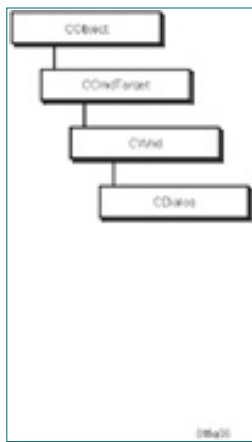
- See “Painting a Window,” p.73

## The Control classes

One of Windows' most identifiable features is its various controls, including buttons, edit boxes, scroll bars, and list boxes. In addition, Windows 95 adds a collection of new, common controls that assist programmers to create professional and powerful applications. These special common controls include animation, progress, slider, spin, and rich text controls. All of these Windows controls are represented by a collection of classes derived from `CWnd`, as shown in Figure 1.14, which lists only some of the most commonly used controls.

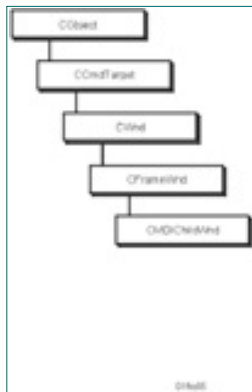
The Windows 95 user interface also features toolbars and status bars, which are encapsulated by MFC's `CToolBar` and `CStatusBar` classes. The `CToolBar` and `CStatusBar` controls are derived from a special control bar class, called, appropriately enough, `CControlBar`. The control bar class hierarchy is shown in Figure 1.15.



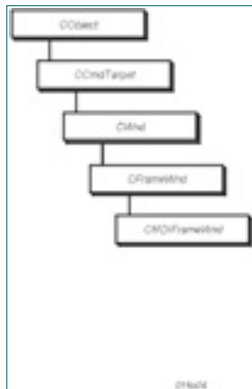


**FIG. 1.13** The drawing-object classes are derived from CGDIObject.

- See “Toolbars and Status Bars,” p. 207



**FIG. 1.14** The control classes are derived from CWnd.



**FIG. 1.15** The control bar classes are derived from CControlBar.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

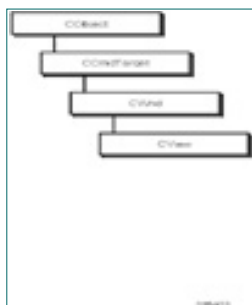
## The *CArchive* and *CFile* Classes

Most applications, no matter what operating system they run under, must save and load data. In the past, writing the code that handles these file tasks took a great deal of time and effort, forcing the programmer to operate at a level much closer to the hardware than was often comfortable. Because the whole point of creating a library of classes is to make things easier for the programmer, MFC doesn't overlook file handling. The *CArchive* and *CFile* classes work together to make the saving and loading of data as painless as possible to perform.

The *CFile* class, directly derived from *CObject* (see Figure 1.16), enables a program to open a file by simply creating a *CFile* object. After creating the *CFile* object, the program can manipulate the file in various ways by calling the *CFile* object's member functions. For example, reading and writing can be performed by calling the *CFile* object's *Read()* and *Write()* member functions. Other member functions enable the program to get the size of the file, get the status of the file, rename the file, seek to a specific position within the file, and more.

Some other interesting classes that can trace their ancestry back to *CFile* are *CInternetFile*, *CGopherFile*, and *CHttpFile*, which, along with Internet classes like *CInternetConnection*, make handling data transfer on the Internet remarkably easy.

Closely related to *CFile* is the *CArchive* class, which encapsulates the serialization of an object's data. To create a *CArchive* object, you must have first created a *CFile* object, which you pass as a parameter to the *CArchive* class's constructor. The *CArchive* class is much simpler than *CFile* in that it provides fewer member functions. In fact, *CArchive*'s member functions do little more than read and write data. *CArchive* is a class unto itself—that is, it is not derived from any other MFC class.



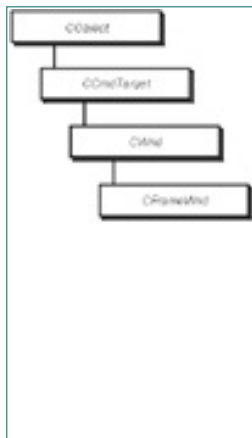
**FIG. 1.16** The *CFile* class is derived from *CObject*.

## The Database Classes

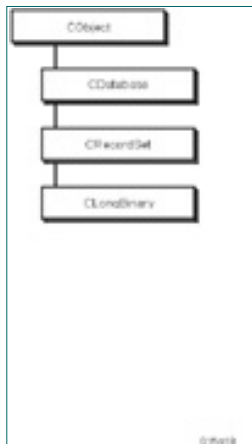
If you've programmed in a corporate environment, you've undoubtedly had to do extensive work with databases. Databases are a corporation's oxygen; without databases, the corporations would not survive. Unfortunately, there are many types of databases and database managers, so it takes much experience to become familiar with the many ways databases can store and manipulate data.

MFC to the rescue! MFC provides ODBC (Open Database Connectivity), as well as DAO (Database Access Object), classes. Both of these class libraries work similarly, enabling you to program front-end database applications without being too concerned with the details of how the database is implemented. The big difference is that the DAO classes rely on Microsoft's Jet database engine, whereas the ODBC classes encapsulate the ODBC API, which processes database access requests through an ODBC driver.

If all of this DAO and ODBC talk sounds like a lot of high tech mumbo-jumbo, you'll be pleased to know that Chapter 22, "Database Programming," deals with these subjects in greater detail. After you've worked through that chapter, you'll be able to dazzle even the big boys with your database prowess. As shown in Figures 1.17 and 1.18, the DAO and ODBC classes are derived from MFC's CObject base class.



**FIG. 1.17** The DAO database classes are derived from CObject.



**FIG. 1.18** The ODBC database classes are also derived from CObject.

## The General Support Classes

MFC features a host of other classes that do everything from handling linked lists to enabling you to create OLE applications. The following list introduces you to some of these many classes:

- Frequently used data types and structures are represented by the CPoint, CRect, CString, CTime, and CTimeSpan classes.
- Thread synchronization is handled by the CCriticalSection, CEvent, CMutex, and CSemaphore classes, all of which are derived from the CSyncObject class.
- Dynamic arrays are represented by the CByteArray, CDWordArray, CObArray, CPtrArray, CStringArray, CUIIntArray, and CWordArray classes.
- Tables can be created using the CMapWordToPtr, CMapPtrToWord, CMapPtrToPtr, CMapWordToOb, CMapStringToPtr, CMapStringToOb, and CMapStringToString classes.

As you can see, MFC provides a class for just about anything you can think of. And, there are many other classes that I haven't even mentioned in this chapter. However, in spite of this slight lack, you should now have a pretty good idea of what MFC can do for you and your programming projects. If you're feeling a bit overwhelmed, you're not alone, I assure you. There's a lot to learn. As you work through this book, you'll become more and more comfortable with MFC's strange ways. Soon, you might wonder how you ever got along without MFC.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Chapter 2

# Using AppWizard to Create an MFC Program

- See how wizards help application users
- Discover how to use AppWizard to create MFC programs
- Understand the classes and files created by AppWizard

At this point, you've got a vague idea of what MFC is, and you're undoubtedly anxious to dig in, get to work, and see what all of this stuff can do for you. What you might not know is that there are two approaches you can take when creating an MFC program. The first approach is to write all of the source code from scratch, just as you're used to doing with your past programs. This first approach is the one you'll be using for most of this book because it gives you a chance to really dig into MFC and see what makes it tick.

The second approach you can take when creating an MFC program is to let Visual C++'s excellent wizards, especially AppWizard, do a lot of the work for you. AppWizard can greatly speed program development by automatically creating a skeleton application that you can build upon to create your own specific application. Because AppWizard can be an important part of using MFC, this chapter, and the next, focuses on the basics of using AppWizard to write a Windows application.

## Understanding Wizards

If you've used any Microsoft products recently, you're probably already familiar with wizards and what they do. Most of Microsoft's products are loaded with wizards, and nothing would make Microsoft happier than to see other companies also feature wizards in their applications. In fact, wizards are a suggested part of a Windows 95 application and are supported by MFC.

If you're not familiar with the term "wizard" yet, a definition is in order. A wizard is simply an automated task, kind of a fancy macro. A Word for Windows user, for example, might use a letter wizard to start a letter. The wizard guides the user, step-by-step, through the process of creating the letter, requesting information in dialog boxes and finally creating a blank letter for the user to fill in with her text.

Visual C++ has its share of wizards, too. The two most important are

AppWizard and ClassWizard. AppWizard guides you through the creation of a “vanilla” MFC application. After you have this skeleton application built, you add your own code to make the application do what you want it to do. ClassWizard, on the other hand, helps you manage the many classes, data members, and member functions in your program. In this chapter, you’ll see how to use AppWizard to build a Windows 95 application. In Chapter 3, “Documents and Views,” you’ll also get some experience with ClassWizard.

## Creating Your First MFC Program

I could spend dozens of pages trying to describe how helpful wizards are. When I was done, you still wouldn’t realize how powerful wizards can be. Seeing is believing, so in this chapter you create an MFC program using AppWizard. The final application will feature all of the goodies you’re used to seeing on a full-fledged Windows 95 application, including a docking toolbar, a status bar, a menu bar, an About dialog box, and more. Just crank up Visual C++, and then follow these steps to create your first MFC program.

1. Select File, New from Developer Studio’s menu bar. The New property sheet appears, as shown in Figure 2.1.



**FIG. 2.1** The New property sheet enables you to select the type of files you want to start.

2. Make sure MFC AppWizard (exe) is selected in the left pane. Then type **app1** into the Project Name box and the path of the folder into which you want the project files stored into the Location box. (If you like, you can use the Browse button to locate the folder into which you want the files stored.)
3. Click the OK button. The MFC AppWizard – Step 1 dialog box appears (see Figure 2.2). On this dialog box, you can choose to create an SDI, MDI, or dialog-based application. You can also choose a language.
4. Leave the default options selected, and click the Next button. The MFC AppWizard – Step 2 of 6 dialog box appears, as shown in Figure 2.3. If you were creating a database application, you could choose the type of database support you need.



**FIG. 2.2** In the Step 1 dialog box, you select the type of application you want to build.



**FIG. 2.3** The Step 2 dialog box enables you to select database support.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

5. Leave the database support set to None, and click the Next button. The MFC AppWizard – Step 3 of 6 dialog box appears, as shown in Figure 2.4. If you were creating an OLE application, you could select OLE support from the given options on this page of the wizard.



**FIG. 2.4** You use the Step 3 dialog box to select OLE support options.

6. Accept the default OLE options (None) by clicking the Next button. The MFC AppWizard – Step 4 of 6 dialog box appears (see Figure 2.5). On this page of the wizard, you select the features you want to include in your application, including a toolbar, a status bar, and printing and messaging abilities.



**FIG. 2.5** You use the Step 4 dialog box to select your application's main features.

7. Leave the default features selected and click the Next button. The MFC AppWizard – Step 5 of 6 dialog box appears (see Figure 2.6). Here, you can choose whether the AppWizard-generated source code will include comments and whether the MFC library should be loaded as a DLL (shared) or linked directly into your application's executable file (static).



**FIG. 2.6** In Step 5, you can choose to include AppWizard-generated comments in the source code, as well as select how your application will use the MFC libraries.

8. Accept the default settings by clicking the Next button. The MFC AppWizard – Step 6 of 6 dialog box appears (see Figure 2.7), which lists the classes that AppWizard is about to create for you. Although you



can change the classes' names, you'll usually leave them as they are.



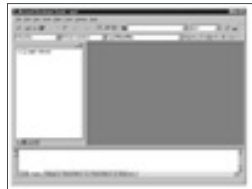
**FIG. 2.7** The Step 6 dialog box gives you a chance to rename classes.

**9.** Leave the classes named as they are, and click the Finish button. The New Project Information dialog box appears, as shown in Figure 2.8. This dialog box displays a summary of the application you've decided to build.



**FIG. 2.8** The New Project Information dialog box summarizes the selections you've made.

**10.** Click the OK button to create your new MFC application. AppWizard generates the source code for the application you selected. When AppWizard is finished, Developer Studio's window will look something like Figure 2.9.



**FIG. 2.9** After MFC creates your application's source code, Developer Studio's main window will display a pane containing your project.

**11.** Click the Build button on the project toolbar or select Build, Build from the menu bar. Developer Studio compiles and links your new MFC application.

## Running Your First MFC Application

And that's all there is to it! After Developer Studio finishes compiling and linking the application, select Build, Execute from the menu bar to run the program. When you do, you see the window shown in Figure 2.10. Pretty darn good looking application for a few mouse clicks, no? Although the preceding steps might have seemed long as you were working through them, you can build the application shown in Figure 2.10 in less than 30 seconds. Wizardry, indeed!



**FIG. 2.10** With AppWizard and a few mouse clicks, you can create an impressive application skeleton.

Except for the fact that your new application doesn't process any kind of data, it's surprisingly complete. You can, for example, create new document windows simply by clicking the toolbar's New button or by selecting File, New from the menu bar. If you select Help, About App1, the application's About dialog box appears. You can (after closing the About dialog box) grab the toolbar with your mouse pointer and drag it somewhere else in the window, changing it from a toolbar to a toolbox. If you select the File, Open command, the Open dialog box appears, enabling you to choose a file to load. Although the file won't actually load, if you select a file, a new document window will appear with the file's name in the document window's title bar. Your new application even features cool stuff like tool tips—those little hint boxes that appear when you leave the mouse pointer over a toolbar button for a second or two.

Go ahead and take a few minutes now to explore your new AppWizard-generated MFC application. Try all of the stuff in the previous paragraph, as well as experiment with other commands like File, Print, and File, Print Preview. And don't forget the commands on the View menu, which enable you to add or remove the control bars, or the handy commands in the Window menu, which arrange the document windows in various ways. When you're done playing, meet me at the next section.

## Exploring AppWizard's Files and Classes

If you take a quick look into your project's folder, you'll discover that AppWizard created a whole slew of source code and data files for your application. What files appear in your project folder depends on the selections you made when you ran AppWizard. If you carefully followed the steps for creating the application, AppWizard generated a folder called RES that contains some of your project's resources, as well as a folder called Debug or Release (depending on whether you're creating a debugging or release version of the program) that contains all of the project's output files. It also generates 18 other files that make up the application's source code. The files AppWizard generates for the app1 project are the following:

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------


[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

BriefFull

- Advanced
- Search
- Search Tips

BROWSE

BY TOPIC

[an error occurred while processing this directive]

## Chapter 3

# Documents and Views

- How document objects declare their data
- How view objects access the document’s data
- How to modify the document and to view classes
- How to save and load a document’s data
- How to edit and display a document’s data

In Chapter 2, “Using AppWizard to Create an MFC Program,” you learned the basics of generating an application with AppWizard. The example application featured all the bells and whistles of a commercial Windows 95 application, including a toolbar, a status bar, tool tips, menus, and even an About dialog box. However, in spite of all those features, the application really didn’t do anything useful. In order to create an application that does more than look pretty on your desktop, you’ve got to modify the code that AppWizard generates. This task can be easy or complex, depending upon how you want your application to look and act.

Before you can perform any modifications, however, you have to know about MFC’s document/view architecture, which is a way to separate an appli-cation’s data from the way the user actually views and manipulates that data. Simply, the document object is responsible for storing, loading, and saving the data, whereas the view object (which is just another type of window) enables the user to see the data on the screen and to edit that data as is appropriate to the application. In the sections that follow, you’ll learn the basics of how MFC’s document/view architecture works.

## Understanding the Document Class

In Chapter 2, you created a basic AppWizard application. When you looked over the various files generated by AppWizard, you discovered a class called CApp1Doc, which was derived from MFC’s CDocument class. In the app1 application, CApp1Doc is the class from which the application instantiates its document object, which is responsible for holding the application’s document data. Because you didn’t modify the AppWizard-generated files, the CApp1Doc class really holds no data. It’s up to you to add storage for the document by adding data members to the CApp1Doc class.

To see how this works, look at Listing 3.1, which shows the header file, AppWizard, created for the CApp1Doc class.

**Listing 3.1 APP1DOC.H—The Header File for the CApp1Doc Class**

```
#if !defined(APP1_H__99206D24_7535_11D0_847F_444553540000__INCLUDED_)
#define APP1_H__99206D24_7535_11D0_847F_444553540000__INCLUDED_

// app1.h : main header file for the APP1 application
//

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"           // main symbols

////////////////////////////////////
```

```

// CApp1App:
// See appl.cpp for the implementation of this class
//

class CApp1App : public CWinApp
{
public:
    CApp1App();

    // Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CApp1App)
    public:
    virtual BOOL InitInstance();
    //}AFX_VIRTUAL

    // Implementation

    //{AFX_MSG(CApp1App)
    afx_msg void OnAppAbout();
    // NOTE - the ClassWizard will add and remove member functions here.
    //      DO NOT EDIT what you see in these blocks of generated code !
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

////////////////////////////////////

//{AFX_INSERT_LOCATION}
// Microsoft Developer Studio will insert additional declarations immediately
// before the previous line.
#endif // !defined(APP1_H__99206D24_7535_11D0_847F_444553540000__INCLUDED)

```

Near the top of Listing 3.1, you can see the class declaration's Attributes section, which is followed by the `public` keyword. This is where you declare the data members that will hold your application's data. In the program you'll create a little later in this chapter, in the section titled "Creating the Rectangles Application," the application must store an array of `CPoint` objects as the application's data. That array is declared as a member of the document class like this:

```

// Attributes
public:
    CPoint points[100];

```

Notice also in the class's header file that the `CApp1Doc` class includes two virtual member functions called `OnNewDocument()` and `Serialize()`. MFC calls the `OnNewDocument()` function whenever the user chooses the File, New command (or its toolbar equivalent if a New button exists). You can use this function to perform whatever initialization must be performed on your document's data. The `Serialize()` member function is where the document class loads and saves its data. When you build this chapter's sample program, you won't believe how easy it is to load and save data.

## Understanding the View Class

As I mentioned previously, the view class is responsible for displaying, and enabling the user to modify, the data stored in the document object. To do this, the view object must be able to obtain a pointer to the document object. After obtaining this pointer, the view object can access the document's data members in order to display or modify them. (Yes, I realize that this sort of interactivity between classes breaks all the object-oriented programming (OOP) rules of data encapsulation, but sometimes you have to give up some things in order to gain others.) If you look at Listing 3.2, you can see how the `CApp1View` class, which you created in Chapter 2, obtains pointers to the document object.

- See "Creating Your First MFC Program," p. 32

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Listing 3.2 APP1VIEW.H—The Header File for the CApp1View Class

```

#if !defined(APP1VIEW_H__99206D2E_7535_11D0_847F_444553540000__INCLUDED_)
#define APP1VIEW_H__99206D2E_7535_11D0_847F_444553540000__INCLUDED_

// app1View.h : interface of the CApp1View class
//
/////////////////////////////////////////////////////////////////

class CApp1View : public CView
{
protected: // create from serialization only
    CApp1View();
    DECLARE_DYNCREATE(CApp1View)

// Attributes
public:
    CApp1Doc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CApp1View)
public:
    virtual void OnDraw(CDC* pDC); // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}AFX_VIRTUAL

// Implementation
public:
    virtual ~CApp1View();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{AFX_MSG(CApp1View)
    // NOTE - the ClassWizard will add and remove member functions here.
    //      DO NOT EDIT what you see in these blocks of generated code !
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

```

#ifndef _DEBUG // debug version in app1View.cpp
inline CApp1Doc* CApp1View::GetDocument()
{ return (CApp1Doc*)m_pDocument; }
#endif

////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately
before the previous line.

#endif // !defined(APP1VIEW_H__99206D2E_7535_11D0_847F_444553540000__INCLUDED)

```

Near the top of Listing 3.2, you can see the class's public attributes, where it declares the `GetDocument()` function as returning a pointer to a `CApp1Doc` object. Anywhere in the view class that you need to access the document's data, you can call `GetDocument()` to obtain a pointer to the document. For example, to add a `CPoint` object to the aforementioned array of `CPoint` objects stored as the document's data, you might use the following line:

```
GetDocument()->m_points[x] = point;
```

You could, of course, do the preceding a little differently by storing the pointer returned by `GetDocument()` in a local pointer variable and then using that pointer variable to access the document's data, like this:

```
pDoc = GetDocument();
pDoc->m_points[x] = point;
```

The second version is more convenient when you need to use the document pointer in several places in the function or when the less clear `GetDocument()->variable` version makes code difficult to understand.

Notice that the view class, like the document class, also overrides a number of virtual functions from its base class. As you'll soon see, the `OnDraw()` function, which is the most important of these virtual functions, is where you paint your window's display. As for the other functions, MFC calls `PreCreateWindow()` before the window element (that is, the actual Windows window) is created and attached to the MFC window class, giving you a chance to modify the window's attributes (such as size and position). Finally, the `OnPreparePrinting()` function enables you to modify the Print dialog box before it's displayed to the user; the `OnBeginPrinting()` function gives you a chance to create GDI objects like pens and brushes that you need to handle the print job; and `OnEndPrinting()` is where you can destroy any objects you may have created in `OnBeginPrinting()`.

---

#### NOTE:

When you first start using an application framework, such as MFC, it's easy to get confused about the difference between an object instantiated from an MFC class and the Windows element it represents. For example, when you create an MFC frame window object, you're actually creating two things: the MFC object that contains functions and data and a Windows window that you can manipulate using the functions of the MFC object. The window element is associated with the MFC class, but it is also an entity unto itself. The window element becomes associated with the window class when MFC calls the `OnCreate()` member function.

---

## Creating the Rectangles Application

Now that you've had an introduction to documents and views, a little hands-on experience should help you better understand how these classes work. In the following steps, you'll build the Rectangles application, which not only demonstrates the manipulation of documents and views, but also gives you a chance to edit an application's resources as well as use ClassWizard to add message response functions to an application.

### Creating the Basic Rectangles Application

Follow the first steps to create the basic Rectangles application and modify its resources:

The complete source code and executable file for the Rectangles application can be found in the CHAP03\Recs directory of this book's CD-ROM.

1. Use AppWizard to create the basic files for the Rectangles program, selecting the options listed in the following table. When you're done, the New Project Information dialog box appears, as shown in Figure 3.1.



Click the OK button to create the project files.



**FIG. 3.1** New Project Information dialog box should look like this.

Dialog Box Name	Options to Select
New Project	Name the project recs, and set the project path to the directory into which you want to store the project's files. Leave the other options set to their defaults.
Step 1 of 6	Select Single Document.
Step 2 of 6	Leave set to defaults.
Step 3 of 6	Leave set to defaults.
Step 4 of 6	Turn off all application features except <u>P</u> rinting and Print Preview.
Step 5 of 6	Leave set to defaults.
Step 6 of 6	Leave set to defaults.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Brief    Full  
 + [Advanced Search](#)  
 + [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

2. Select the ResourceView tab in the project workspace window. Visual C++ displays the ResourceView window, as shown in Figure 3.2.



**FIG. 3.2** The ResourceView tab displays the ResourceView window.

3. In the ResourceView window, click the plus sign next to `recs` resources to display the application's resources. Click the plus sign next to `Menu`, and then double-click the `IDR_MAINFRAME` menu ID. The Visual C++ menu editor appears, as shown in Figure 3.3.



**FIG. 3.3** The menu editor enables you to modify the application's default menu bar.

4. Click the Rectangles application's Edit menu (not the Visual C++ Edit menu), and then press your keyboard's Delete key to delete the Edit menu. When you do, a dialog box asks for verification of the delete command (see Figure 3.4). Click the OK button.



**FIG. 3.4** The menu editor enables you to delete items from the menus.

5. Double-click the About `recs...` item in the Help menu, and change it to About Rectangles... (by typing **&About Rectangles&** in the Caption box), as shown in Figure 3.5. Close the menu editor.



**FIG. 3.5** You should change the About command to include the

application's actual name.

6. Double-click the Accelerator resource in the ResourceView window. Double-click the IDR\_MAINFRAME accelerator ID to bring up the accelerator editor, as shown in Figure 3.6.



**FIG. 3.6** The accelerator editor enables you to edit your application's hotkeys.

7. Using your keyboard's arrow and Delete keys, delete all accelerators except ID\_FILE\_NEW, ID\_FILE\_OPEN, ID\_FILE\_PRINT, and ID\_FILE\_SAVE (see Figure 3.7). Close the accelerator editor.



**FIG. 3.7** You can use the accelerator editor to delete unwanted hotkeys.

8. Double-click the Dialog resource in the ResourceView window. Double-click the IDD\_ABOUTBOX dialog box ID to bring up the dialog box editor (see Figure 3.8).



**FIG. 3.8** In the dialog box editor, you can modify your application's About dialog box.

9. Modify the dialog box by changing the title to "About Rectangles," changing the first static text string to Rectangles, Version 1.0, and adding the static string by Macmillan Computer Publishing, as shown in Figure 3.9. Close the dialog box editor.



**FIG. 3.9** The About Rectangles dialog box should end up like this.

10. Double-click the String Table resource in the ResourceView window. Double-click the String Table ID to bring up the string table editor, as shown in Figure 3.10.



**FIG. 3.10** The string table holds the various messages that appear in your application's windows.

11. Double-click the IDR\_MAINFRAME string, and then change the first segment of the string to Rectangles, as shown in Figure 3.11. Close the string table editor.



**FIG. 3.11** The first segment of the IDR\_MAINFRAME string appears in your main window's title bar.

## Modifying the Document Class

Now that you have the application's resources the way you want them, it's time to add code to the document and to the view classes in order to create an application that actually does something. Perform the following steps to add the code that modifies the document class to handle the application's data (which is an array of CPoint objects that determines where rectangles should be drawn in the view window):

1. Click the FileView tab to display the FileView window. Then, click the plus sign next to the folder labeled "recs files" to display the project's file folders, as shown in Figure 3.12.



**FIG. 3.12** The FileView window lists folders for the source files that make up your project.

2. Double-click the Header Files folder, and then double-click the recsDoc.h entry in the file list. The recsDoc.h file appears in the code window, as shown in Figure 3.13.



**FIG. 3.13** By clicking entries in the FileView window, you can load and display source code files in the code window.

3. Add the following lines to the attributes section of the CRecsDoc class

right after the public keyword:

```
CPoint m_points[100];  
UINT m_pointIndex;
```

These lines declare the data members of the document class, which will store the application's data. In the case of the Rectangles application, the data in the `m_points[]` array represents the locations of rectangles displayed in the view window. The `m_pointIndex` data member holds the index of the next empty element of the array.

4. Double-click Source Files in the FileView window, and then double-click `recsDoc.cpp` to open the document class's implementation file. Add the following line to the `OnNewDocument()` function right after the (SDI documents will reuse this document) comment:

```
m_pointIndex = 0;
```

This line initializes the index variable each time a new document is started, ensuring that it always starts off by indexing the first element in the `m_points[]` array.

5. Add the lines shown in Listing 3.3 to the `Serialize()` function right after the `TODO: add storing code here` comment.

---

**Listing 3.3 LST03\_03.CPP—Code for Saving the Document's Data**

---

```
ar << m_pointIndex;  
  
for (UINT i=0; i<m_pointIndex; ++ i)  
{  
    ar << m_points[i].x;  
    ar << m_points[i].y;  
}
```

---

This is the code that saves the document's data. As you can see, you have to do nothing more than use the `<<` operator to direct the data to the archive object. You'll look at this code in more detail later in this chapter, in the section titled "Serializing Document Data."

6. Add the lines shown in Listing 3.4 to the `Serialize()` function right after the `TODO: add loading code here` comment.

---

**Listing 3.4 LST03\_04.CPP—Code for Loading the Document's Data**

---

```
ar >> m_pointIndex;  
  
for (UINT i=0; i<m_pointIndex; ++ i)  
{  
    ar >> m_points[i].x;  
    ar >> m_points[i].y;
```

```
}
```

```
UpdateAllViews(NULL);
```

---

This is the code that loads the document's data. You need only use the >> operator to direct the data from the archive object into the document's data storage. You'll look at this code in more detail later in this chapter, in the section titled "Serializing Document Data."

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Modifying the View Class

This finishes the modifications you must make to the document class. In the following steps, you make the appropriate changes to the view class, enabling the class to display, modify, and print the data stored in the document class:

1. Load the recsView.cpp file, and add the lines shown in Listing 3.5 to the OnDraw() function right after the TODO: add draw code for native data here comment.

### Listing 3.5 LST03\_05.CPP—Code for Displaying the Application’s Data (Rectangles)

```
UINT pointIndex = pDoc->m_pointIndex;

for (UINT i=0; i<pointIndex; ++i)
{
    UINT x = pDoc->m_points[i].x;
    UINT y = pDoc->m_points[i].y;
    pDC->Rectangle(x, y, x+20, y+20);
}
```

#### Note:

The code in Listing 3.5, which iterates through the document object’s m\_points[] array and displays rectangles at the coordinates it finds in the array, is executed whenever the application’s window needs repainting.

2. Add the following line to the very beginning of the OnPreparePrinting() function:

```
pInfo->SetMaxPage(1);
```

This line modifies the common Print dialog box such that the user cannot try to print more than one page.

3. Choose View, ClassWizard. The ClassWizard property sheet appears, as shown in Figure 3.14.



**FIG. 3.14** You can use ClassWizard to add functions to a class.

4. Make sure that CRecsView is selected in the Class Name and Object IDs boxes. Then, double-click WM\_LBUTTONDOWN in the Messages box to add the OnLButtonDown() message response function to the class, as shown in Figure 3.15.



**FIG. 3.15** ClassWizard takes all the work out of creating response functions for Windows messages.

The OnLButtonDown() function is now associated with Windows' WM\_LBUTTONDOWN message, which means MFC will call OnLButtonDown() whenever the application receives a WM\_LBUTTONDOWN message.

5. Click the Edit Code button to jump to the OnLButtonDown() function in your code. Then, add the lines shown in Listing 3.6 to the function right after the TODO: Add your message handler code here and/or call default comment.

**Listing 3.6 LST03\_06.CPP—Code to Handle Left Button Clicks**

---

```
CRecsDoc *pDoc = GetDocument();

if (pDoc->m_pointIndex == 100)
    return;

pDoc->m_points[pDoc->m_pointIndex] = point;
++pDoc->m_pointIndex;
pDoc->SetModifiedFlag();
Invalidate();
```

---

**Note:**

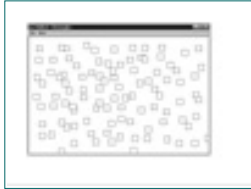
The code in Listing 3.6 adds a point to the document's point array each time the user clicks the left mouse button over the view window. The call to Invalidate() causes MFC to call the OnDraw() function, where the window's display is redrawn with the new data.

---

You've now finished the complete application. Click the Build button on the toolbar or choose the Build, Build command from the menu bar to compile and link the application.

## Running the Rectangles Application

Once you have the Rectangles application compiled and linked, run it by choosing Build, Execute. When you do, you'll see the application's main window. Place your mouse pointer over the window's client area and left click. A rectangle appears. Go ahead and keep clicking. You can place up to 100 rectangles in the window (see Figure 3.16).



**FIG. 3.16** The Rectangles application's document consists of rectangles that the user places on the screen.

To save your work (this is work?), choose the File, Save. You can view your document in print preview by choosing File, Print Preview, or just go ahead and print by choosing File, Print. Of course, you can create a new document by choosing File, New, or load a document you previously saved by choosing File, Open. Finally, if you choose Help, About Rectangles, you'll see the application's About dialog box (see Figure 3.17).



**FIG. 3.17** The About Rectangles dialog box provides information about the application.

## Exploring the Rectangles Application

If this is your first experience with AppWizard and MFC, you're probably amazed at how much you can do with a few mouse clicks and a couple of dozen lines of code. You're also probably still a little fuzzy on how the program actually works, so in the following sections, you'll examine the key parts of the Rectangles application. Keep in mind that what you learn here is only the first step toward understanding MFC. Only the primary issues of using AppWizard and MFC are covered in the following section. Subsequent chapters in this book solve many of the remaining MFC mysteries.

### Declaring Storage for Document Data

As you've read again and again since the beginning of this book, it is the document object in an AppWizard-generated MFC program that is responsible for maintaining the data that makes up the application's document. For a word processor, this data would be strings of text, whereas for a paint program, this data might be a bitmap. For the Rectangles application, the document's data are the coordinates of rectangles displayed in the view window.



The first step in customizing the document class, then, is to provide the storage you need for your application's data. How you do this, of course, depends on the type of data you must use. But, in every case, the variables that will hold that data should be declared as data members of the document class, as is done in the Rectangles application. Listing 3.7 shows the relevant code.

### **Listing 3.7 LST03\_07.CPP—Declaring the Document Data of the Rectangles Application**

---

```
// Attributes
public:

    CPoint m_points[100];
    UINT m_pointIndex;
```

---

In Listing 3.7, the document-data variables `m_points[]` and `m_pointIndex` are declared as public members of the document class. (The `m` prefix indicates that the variables are members of the class, rather than global or local variables. This is a tradition that Microsoft started. You can choose to follow it or not.) The `m_points[]` array holds the coordinates of the rectangles displayed in the view window, and the `m_pointIndex` variable holds the number of the next empty element in the array. You can also think of `m_pointIndex` as the current rectangle count. These variables are declared as public so that the view class can access them. If you were to declare the data variables as protected or private, your compiler would whine loudly when you tried to access the variables from your view class's member functions.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The data storage for the Rectangles application is pretty trivial in nature. For a commercial-grade application, you'd almost certainly need to keep track of much more complex types of data. But the method of declaring storage for the document would be the same. You may, of course, also declare data members that you use only internally in the document class—variables that have little or nothing to do with the application's actual document data. However, you should declare such data members as protected or private.

## Initializing Document Data

Once you have your document's data declared, you usually need to initialize it in some way each time a new document is created. For example, in the Rectangles application, the `m_pointIndex` variable must be initialized to zero when a new document is started. Otherwise, the `m_pointIndex` may contain an old value from a previous document, which could make correctly accessing the `m_points[]` array as tough as getting free cash from an ATM. In the Rectangles application, `m_pointIndex` gets initialized in the `OnNewDocument()` member function, as shown in Listing 3.8.

### Listing 3.8 LST03\_08.CPP—The Rectangles Application’s OnNewDocument() Function

[illegible]

```

////////////////////////////////////
////////////////////////////////////

return TRUE;
}

```

---

#### Note:

In many of the listings in this chapter, code that you added to a function is located between START CUSTOM CODE and END CUSTOM CODE comment blocks. That is, all of the code in Listing 3.8, except the line `m_pointIndex = 0;` was created by AppWizard.

---

MFC calls the `OnNewDocument()` function whenever the user starts a new document, usually by choosing File, New. As you can see, `OnNewDocument()` first calls the base class's `OnNewDocument()`, which calls `DeleteContents()` and then marks the new document as *clean* (meaning it doesn't yet need to be saved due to changes).

What's `DeleteContents()`? It's another virtual member function of the `CDocument` class. If you want to be able to delete the contents of a document without actually destroying the document object, you can override `DeleteContents()` to handle this task.

Keep in mind that how you use `OnNewDocument()` and `DeleteContents()` depends on whether you're writing an SDI or MDI application. In an SDI application, the `OnNewDocument()` function indirectly destroys the current document by reinitializing it in preparation for new data. An SDI application, after all, can contain only a single document at a time. In an MDI application, `OnNewDocument()` simply creates a brand new document object, leaving the old one alone. For this reason, you can perform general document initialization in the class's constructor in an MDI application.

## Serializing Document Data

If your application is going to be useful, it must be able to do more than display data; it must also be able to load and save data sets created by the user. Writing this book would have been a nightmare if my text disappeared every time I shut down my word processor! The act of loading and saving document data with MFC is called *serialization*. In spite of the complications you may have experienced with files in the past, loading and saving data with MFC is a snap, thanks to the `CArchive` class (an object of which is passed to the document class's `Serialize()` member function). Listing 3.9 shows the Rectangles application's `Serialize()` function.

### Listing 3.9 LST03\_09.CPP—The Rectangles Application's `Serialize()` Function

---

```

void CRecsDoc::Serialize(CArchive& ar)
{

```

```

if (ar.IsStoring())
{
    // TODO: add storing code here

    //////////////////////////////////////////
    //////////////////////////////////////////
    // START CUSTOM CODE
    //////////////////////////////////////////
    //////////////////////////////////////////

    ar << m_pointIndex;

    for (UINT i=0; i<m_pointIndex; ++ i)
    {
        ar << m_points[i].x;
        ar << m_points[i].y;
    }

    //////////////////////////////////////////
    //////////////////////////////////////////
    // END CUSTOM CODE
    //////////////////////////////////////////
    //////////////////////////////////////////
}
else
{
    // TODO: add loading code here

    //////////////////////////////////////////
    //////////////////////////////////////////
    // START CUSTOM CODE
    //////////////////////////////////////////
    //////////////////////////////////////////

    ar >> m_pointIndex;

    for (UINT i=0; i<m_pointIndex; ++ i)
    {
        ar >> m_points[i].x;
        ar >> m_points[i].y;
    }

    UpdateAllViews(NULL);

    //////////////////////////////////////////
    //////////////////////////////////////////
    // END CUSTOM CODE
    //////////////////////////////////////////
    //////////////////////////////////////////
}

```

}

As you can see in Listing 3.9, the `Serialize()` function receives a reference to a `CArchive` object as its single parameter. At this point, MFC has done all the file-opening work for you. All you have to do is use the `CArchive` object to load or save your data. How do you know which to do? MFC has already created the lines that call the `CArchive` object's `IsStoring()` member function, which returns `TRUE` if you need to save data and `FALSE` if you need to load data.

Thanks to the overloaded `<<` and `>>` operators in the `CArchive` class, you can save and load data exactly as you're used to doing, using C++ input/output objects. If you look at the `Serialize()` function, you'll notice that about the only difference between the saving and loading of data is the operator that's used. One other difference is the call to `UpdateAllViews()` after loading data. `UpdateAllViews()` is the member function that notifies all views attached to this document that they need to redraw their data displays. When calling `UpdateAllViews()`, you almost always use `NULL` as the single parameter. If you should ever call `UpdateAllViews()` from your view class, you should send a pointer to the view as the parameter.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Displaying Document Data

Now that you've got your document class ready to store, save, and load its data, you need to customize the view class so that it can display the document data, as well as enable the user to modify the data. In an MFC application using the document/view model, it's the OnDraw() member function of the view class that is responsible for displaying data, either on the screen or the printer. Listing 3.10 shows the Rectangles application's version of OnDraw().

### Listing 3.10 LST03\_10.CPP—The Rectangles Application's OnDraw() Function

```

void CRecsView::OnDraw(CDC* pDC)
{
    CRecsDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here

    //////////////////////////////////////
    //////////////////////////////////////
    // START CUSTOM CODE
    //////////////////////////////////////
    //////////////////////////////////////

    UINT pointIndex = pDoc->m_pointIndex;

    for (UINT i=0; i<pointIndex; ++i)
    {
        UINT x = pDoc->m_points[i].x;
        UINT y = pDoc->m_points[i].y;
        pDC->Rectangle(x, y, x+20, y+20);
    }

    //////////////////////////////////////
    //////////////////////////////////////
    // END CUSTOM CODE
    //////////////////////////////////////
    //////////////////////////////////////
}
    
```

The first thing you should notice about OnDraw() is that its single parameter is a pointer to a CDC object. A CDC object encapsulates a Windows device context, automatically initializing the DC and providing many member functions with which you can draw your application's display. Because the OnDraw() function is responsible for updating the window's display, it's a nice convenience to have a CDC object all ready to go.

- See “Exploring the Paint1 Application,” p. 81

Also notice that—because an application that uses the document/view model stores its data in the document class—AppWizard has generously supplied the code needed to obtain a pointer to that class. In the custom code in OnDraw(), the function uses this document pointer to retrieve the value of the document’s index variable (the number of rectangles currently displayed), and then uses it as a loop control variable. The loop simply iterates through the document’s m\_points[] array, drawing rectangles at the coordinates contained in the CPoint objects stored in the array.

## Modifying Document Data

The view object is not only responsible for displaying the application’s document data; it must also (if appropriate) enable the user to edit that data. Exactly how you enable the user to edit an application’s data depends a great deal upon the type of application you’re building. The possibilities are endless. In the simple rectangles application, the user can edit a document only by clicking in the view window, which adds another rectangle to the document. This happens in response to the WM\_LBUTTONDOWN message, which Windows sends the application every time the user clicks the left mouse button when the mouse pointer is over the view window.

If you recall, you used ClassWizard to add the OnLButtonDown() function to the program. This is the function that MFC calls whenever the window receives a WM\_LBUTTONDOWN message. It is in OnLButtonDown(), then, that the Rectangles application must modify its list of rectangles, adding the new rectangle at the window position the user clicked. Listing 3.11 shows the OnLButtonDown() function where this data update occurs.

### Listing 3.11 LST03\_11.CPP—The Rectangles Application’s OnLButtonDown() Function

---

```
void CRecsView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    //////////////////////////////////////
    //////////////////////////////////////
    // START CUSTOM CODE
    //////////////////////////////////////
    //////////////////////////////////////

    CRecsDoc *pDoc = GetDocument();

    if (pDoc->m_pointIndex == 100)
        return;

    pDoc->m_points[pDoc->m_pointIndex] = point;
    ++pDoc->m_pointIndex;
    pDoc->SetModifiedFlag();
    Invalidate();

    //////////////////////////////////////
    //////////////////////////////////////
    // END CUSTOM CODE
    //////////////////////////////////////
    //////////////////////////////////////

    CView::OnLButtonDown(nFlags, point);
}
```

Of the two parameters received by OnLButtonDown(), it is point that is most useful to the Rectangles application because this CPoint object contains the coordinates at which the user just clicked. In the custom code you added to OnLButtonDown(), the function first obtains a pointer to the document object. Then, if the document object's m\_pointIndex data member is equal to 100, there is no room for another rectangle. In this case, the function immediately returns, effectively ignoring the user's request to modify the document. Otherwise, the function adds the new point to the m\_points[] array and increments the m\_pointIndex variable.

Now that the document's data has been updated as per the user's modification, the document must be marked as *dirty* (needing saving), and the view must display the new data. A call to the document object's SetModifiedFlag() function takes care of the first task. If the user now tries to exit the program without saving the data or tries to start a new document, MFC displays a dialog box warning the user of possible data loss. When the user saves the document's data, the document is set back to *clean*. The call to Invalidate() notifies the view window that it needs repainting, which results in MFC calling the view object's OnDraw() function.

In this chapter, you got a quick look at how an AppWizard-generated application uses MFC to coordinate an application's document and view objects. There is, of course, a great deal more to learn about MFC before you can create your own sophisticated Windows 95 applications. Because AppWizard can disguise much of what is going on in an MFC application, the rest of this book concentrates on writing MFC applications without AppWizard's help. However, most of what you'll learn in the upcoming chapters can be applied to AppWizard-generated applications, as well. Remember that although AppWizard is a useful tool, there's really nothing magical about it: It creates the same type of MFC code that you can create on your own.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 4

# Constructing an MFC Program from Scratch

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

- Creating an application class
- Creating a frame-window class
- Creating a project workspace for your MFC application
- Associating Windows messages with message response functions

AppWizard is great for getting an application started quickly. However, because AppWizard does so much of the work for you, you never get to learn many of the details of MFC programming. Knowing how MFC really works not only enables you to forgo AppWizard when you'd rather do the work yourself, but it also helps you understand AppWizard programs better so you can customize them more effectively. For these reasons, some of this book is dedicated to writing MFC programs without the help of AppWizard. This chapter is your first step toward that goal.

## Writing the Minimum MFC Program

In Chapter 1, “An Introduction to MFC,” you get a quick overview of the many classes that you can use to create your MFC-based applications. From the brief descriptions included in that chapter, you already might have figured out that an MFC program must have at least two classes: the application class and the window class. To get a window up on the screen, you need do very little with these two classes. In fact, you can create a minimal MFC program in fewer lines of code than you'd believe. The following list outlines the steps you must take to create the smallest, functional MFC program:

1. Create an application class derived from CWinApp.
2. Create a window class derived from CFrameWnd.
3. Instantiate a global instance of your application object.
4. Instantiate your main window object in the application object's InitInstance() function.
5. Create the window element (the Windows window) associated with the window object by calling Create() from the window object's constructor.

That's all there is to getting a window up on the screen. In the following sections, you look at these steps in greater detail, examining the actual code that performs the listed tasks.

## Creating the Application Class

As you learned in the previous section, the first step in writing an MFC program is to create your application class, in which you must override the `InitInstance()` function. Listing 4.1 shows the header file for your first handwritten application class, called `CMFCApp1`.

### Listing 4.1 MFCAPP1.H—The Header File for the `CMFCApp1` Class

---

```
////////////////////////////////////  
// MFCAPP1.H: Header file for the CMFCApp1 class, which  
//           represents the application object.  
////////////////////////////////////  
  
class CMFCApp1 : public CWinApp  
{  
public:  
    CMFCApp1();  
    BOOL InitInstance();  
};
```

---

In the first line of the preceding code (not counting the comments), you can see that the `CMFCApp1` class is derived from MFC's `CWinApp` class. The `CMFCApp1` class has a constructor and overrides the `InitInstance()` function, which is the function MFC calls to create the application's window object. Because the base class's `InitInstance()` does nothing but return a value of `TRUE`, if you want your application to have a window, you must override `InitInstance()` in your application class.

To see how the application class actually creates its main window object, take a look at Listing 4.2, which is the `CMFCApp1` class's implementation file.

### Listing 4.2 MFCAPP1.CPP—The Implementation File for the `CMFCApp1` Class

---

```
////////////////////////////////////  
// MFCAPP1.CPP: Implementation file for the CMFCApp1 class,  
//           which represents the application object.  
////////////////////////////////////  
  
#include <afxwin.h>  
#include "mfcapp1.h"  
#include "mainfrm.h"  
  
CMFCApp1 MFCApp1;  
  
CMFCApp1::CMFCApp1()  
{  
}  
  
BOOL CMFCApp1::InitInstance()
```

```
{  
    m_pMainWnd = new CMainFrame();  
    m_pMainWnd->ShowWindow(m_nCmdShow);  
  
    return TRUE;  
}
```

---

The first thing to notice here is the line

```
#include <afxwin.h>
```

near the top of the file. AFXWIN.H is the header file for the MFC classes, so it must be included in any file that references those classes. Failure to include AFXWIN.H in an MFC program causes the compiler to crank out a long list of error messages, because it won't recognize the MFC classes you're using.

After the include statements, you can see where the program instantiates its global application object with the following line:

```
CMFCApp1 MFCApp1;
```

The application object must be global to ensure that it is the first object instantiated in the program.

In the CMFCApp1 class, the constructor does nothing, although you can add initialization code to the constructor as needed. In this first program, it's the application class's InitInstance() function that's most important. MFC calls InitInstance() once, at the beginning of the application's run. InitInstance()'s main task is to instantiate the application's main window object, which the CMFCApp1 class does with the following line:

```
m_pMainWnd = new CMainFrame();
```

This line creates a new CMainFrame() object on the heap, calling the CMainFrame() class's constructor, where, as you'll soon see, the class creates the main window element. A pointer to the new window object gets stored in m\_pMainWnd, which is a data member of the application class. You must be sure to initialize m\_pMainWnd properly, as it's through this pointer that the application class maintains contact with its main window.

After instantiating the window object, the window element must usually be displayed. This is done by calling the window object's ShowWindow() function, like this:

```
m_pMainWnd->ShowWindow(m_nCmdShow);
```

ShowWindow()'s single argument indicates how the window should be displayed initially. You need only pass along the application object's m\_nCmdShow data member, which contains the value of the nCmdShow parameter that Windows passed to WinMain().

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full
 

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

**Note:**

When an application starts up, there are several ways that its window can be initially displayed. These different display styles are represented by a set of constants defined by Windows, including SW\_HIDE, SW\_MINIMIZE, SW\_SHOWMAXIMIZED, SW\_SHOWMINIMIZED, and SW\_SHOWNORMAL, among others. Luckily, with MFC, you don't need to worry about these different styles unless you want to do something special with a window. Just passing m\_nCmdShow to ShowWindow() as the display style shows the window in the default manner.

**Creating the Frame-Window Class**

Now that you know how to create your application class, which is responsible for instantiating your main window, it'd be nice to see how the window class works. Listing 4.3 is the header file for the CMainFrame class, which represents the CMFCApp1 class's main window.

**Listing 4.3 MAINFRM.H—The Header File for the CMainFrame Class**

```

////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//             represents the application's main window.
////////////////////////////////////

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();
    ~CMainFrame();
};
    
```

As you can see from the class's declaration, CMainFrame couldn't get much simpler, containing only a constructor and a destructor. Of course, CMainFrame is backed with all the power of MFC's CFrameWnd class, from which it's derived. Although the CMainFrame class's header file is interestingly stark, it doesn't provide too many clues about how the application's main window is created. That information lies in the class's implementation file, which is shown in Listing 4.4.

**Listing 4.4 MAINFRM.CPP—The Implementation File for the CMainFrame Class**

```

////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//             class, which represents the application's
//             main window.
////////////////////////////////////

#include <afxwin.h>
#include "mainfrm.h"
    
```

```

CMainFrame::CMainFrame()
{
    Create(NULL, "MFC Appl");
}

CMainFrame::~CMainFrame()
{
}

```

---

If you turn your attention to the class's constructor, you can see that the application's window element is created by a simple call to the window class's `Create()` member function, like this:

```
Create(NULL, "MFC Appl");
```

Actually, this call to `Create()` is deceptively simple. `Create()` really requires eight arguments, although all but the first two have default values. `Create()`'s prototype looks like Listing 4.5.

---

#### **Listing 4.5 LST04\_05.CPP—The `Create()` Function's Prototype**

---

```

BOOL Create(LPCTSTR lpszClassName,
            LPCTSTR lpszWindowName,
            DWORD dwStyle = WS_OVERLAPPEDWINDOW,
            const RECT& rect = rectDefault,
            CWnd* pParentWnd = NULL,          // != NULL for popups
            LPCTSTR lpszMenuName = NULL,
            DWORD dwExStyle = 0,
            CCreateContext* pContext = NULL);

```

---

`Create()`'s arguments are a pointer to the following:

- Class's name
- Pointer to the window's title
- The window's style flags
- The window's size and position (stored in a `RECT` structure)
- Pointer to the parent window
- Pointer to the window's menu name
- Any extended style attributes
- Pointer to a `CCreateContext` structure (which helps MFC manage the document and view objects)

If you use `NULL` for the class name, MFC uses its default window class. As for the title string, this is the title that appears in the window's title bar. If you want to create different types of windows, you can use different style flags for `dwStyle`. For example, the line

```
Create(NULL, "MFC Appl", WS_OVERLAPPED | WS_BORDER | WS_SYSMENU);
```

creates a window with a system menu, a Close button, and a thin border. Because such a window is missing the Minimize and Maximize buttons, as well as the usual thick border, the user cannot change the size of the window but can only move or close it. The styles you can use include `WS_BORDER`, `WS_CAPTION`, `WS_CHILD`, `WS_MAXIMIZEBOX`, `WS_MINIMIZEBOX`, `WS_THICKFRAME`, and more. Please refer to your Windows programming documentation for more information.

By changing the default values of the fourth argument, you can position and size a window. For example, the line

```
Create(NULL, "MFC App1", WS_OVERLAPPEDWINDOW, CRect(0, 0, 100, 100));
```

creates a  $100 \times 100$  window located in the desktop's upper-left corner.

Your main window doesn't have a parent window, so the fifth argument should be NULL. As for the sixth argument, you'll learn about menus in Chapter 6, "Using Menus." In most cases, you won't need to worry about the last two arguments, `dwExStyle` and `pContext`.

## Compiling an MFC Program

Creating your application and window classes is all well and good. However, because you haven't used AppWizard to start your new project, you have to manually create a project for your files so that Developer Studio can compile and link the files into your final executable application. In this section, you create a new project for the MFC App1 application that you just examined. To create a project for the application, perform the easy steps that follow.



The complete source code and executable file for the MFC App1 application can be found in the CHAP04\MFCAPP1 directory of this book's CD-ROM.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

1. Select File, New from Developer Studio's menu bar. The New property sheet appears.
2. If necessary, select the Projects tab. The Projects page appears (see Figure 4.1).



**FIG. 4.1** Selecting the Projects tab brings up the Projects page.

3. Select Win32 Application in the left hand pane, type **mfcapp1** in the Project Name box, set the Location box to the folder in which you want to create the project, and click the OK button. Developer Studio creates the new project workspace.
4. Copy the MFCAPP1.H, MFCAPP1.CPP, MAINFRM.H, and MAINFRM.CPP files from the CHAP04\MFCAPP1 folder on this book's CD-ROM into the project folder that you created in Step 3. (Of course, if you were actually creating this application from scratch, you'd have to create the application's source code files by typing them into the editor.)
5. Select Project, Add to Project, Files from Developer Studio's menu bar. The Insert Files into Project dialog box appears, as shown in Figure 4.2.



**FIG. 4.2** When you select Insert, Files into Project, the Insert Files into Project dialog box appears.

6. Click on MFCAPP1.CPP and MAINFRM.CPP to highlight them (hold down the Ctrl key), and then click the OK button. Developer Studio adds these two source code files to the project.
7. Select the Project, Settings command from the menu bar. The Project Settings property sheet appears, as shown in Figure 4.3.





**FIG. 4.3** If you select Project, Settings, the Project Settings property sheet appears.

8. Change the Microsoft Foundation Classes box to Use MFC in A Shared DLL, and then click the OK button. Developer Studio adds MFC support to the project. (You can also select Use MFC in A Static Library if you want the MFC code to be linked into your application's executable file. This makes your executable file larger but eliminates the need to distribute the MFC DLLs with your application. In most cases, the shared DLL is preferable.)

At this point, you're ready to compile and link your new project. To do this, just click the Build button on the toolbar or select Build, Build from the menu bar. When the project has been compiled and linked, select Build, Execute to run the application. When you do, you see the main window screen. Your first handwritten MFC application might not be fancy, but it's up on the screen!

## Responding to Windows Messages

To do something useful, your new MFC application must be able to respond to messages sent to its window. This is how a Windows application knows what the user and the system are doing. To respond to Windows messages, an MFC application must have a message map that tells MFC what functions handle what messages. In this section, you'll add message mapping to the MFC application.

### Declaring a Message Map

Any class that has `CCmdTarget` as an ancestor can respond to Windows messages. These include application, framewindow, viewwindow, document, and control classes. Where you decide to intercept a message depends a great deal upon your program's design. However, as you discover in Chapter 3, "Documents and Views," the document object often gets to handle messages generated from the File menu, whereas the view object takes control of messages needed to edit and display the document's data.

In your new MFC application, you have no File menu to worry about. Moreover, you don't have a document or view class. So, as is typical in a case like this, you'll add message mapping to the frame window, represented by the `CMainFrame` class.

The first step in adding message mapping is to declare a message map in the class's header file. Placing the line

```
DECLARE_MESSAGE_MAP ( )
```

at the end of the class's declaration, just before the closing brace, takes care of

this easy task. `DECLARE_MESSAGE_MAP()` is a macro defined by MFC. This macro takes care of all the details required to declare a message map for a class. You can look for `DECLARE_MESSAGE_MAP()`'s definition in MFC's source code if you want a better look, but you really don't need to know how it works. Just place the preceding code line in your class's declaration and whisper a quiet thank-you to Microsoft's programmers for making message maps so easy to declare.

## Defining a Message Map

After you have declared your message map in your class's header, it's time to define the message map in your class's implementation file. Usually, programmers place the message map definition near the top of the file, right after the include lines. However, you can place it just about anywhere in the file—as long as it's not nested inside a function or other structure.

What does a message map definition look like? The one that follows maps `WM_LBUTTONDOWN` messages to a message map function called `OnLButtonDown()`:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_LBUTTONDOWN( )
END_MESSAGE_MAP( )
```

The first line in the preceding code is a macro defined by MFC that takes care of initializing the message map definition. The macro's two arguments are the class for which you're defining the message map and that class's immediate base class. You need to tell MFC about the base class so that it can pass any unhandled messages on up the class hierarchy, where another message map might contain an entry for the message.

After the starting macro, you place the message map entries, which match messages with their message map functions. How you do this depends upon the type of message you're trying to map. For example, MFC has previously defined message map functions for all of the Windows system messages, such as `WM_LBUTTONDOWN`. To write a message map for such a message, you simply add an `ON_` prefix to the message name and tack on the parentheses. So `WM_LBUTTONDOWN` becomes `ON_WM_LBUTTONDOWN()`; `WM_MOUSEMOVE` becomes `ON_WM_MOUSEMOVE()`; `WM_KEYDOWN` becomes `ON_WM_KEYDOWN()`, and so on. (Notice that you don't place semicolons after message map entries.)

To determine the matching message map function for the map entry, you throw away the `_WM_` in the macro name and then spell the function name in uppercase and lowercase. For example, `ON_WM_LBUTTONDOWN()` becomes `OnLButtonDown()`; `ON_WM_MOUSEMOVE()` ends up as `OnMouseMove()`; and `ON_WM_KEYDOWN()` matches up with `OnKeyDown()`.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)

 **BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

There are several other types of macros that you can use to define message map entries. For example, as you'll learn in Chapter 6, "Using Menus," the `ON_COMMAND()` macro maps menu commands to their response functions, whereas the `ON_UPDATE_COMMAND_UI()` macro enables you to attach menu items to the functions that keep the menu items' appearance updated (checked, enabled, and so on).

- See "Exploring the Menu Application," p. 102

---

**Note:**

Throughout this chapter, I refer to *message response* functions and *message map* functions. These types of functions are really exactly the same thing. Depending upon context, one term is sometimes more descriptive than the other. Microsoft sometimes refers to these functions as *message handlers*.

---

## Writing Message Map Functions

Now that you have your message map declared and defined, you're missing only one important element: the message map functions themselves! In the example message map, there's only a single message map entry, which is `ON_WM_LBUTTONDOWN()`. As you now know, this is the macro for the `WM_LBUTTONDOWN` Windows message. In order to complete the message mapping, you must now write the matching `OnLButtonDown()` function.

First, you need to declare the message map functions in your class's header file. And here's where you hit your first stumbling block. How does MFC prototype the various message map functions? Here's a neat trick. You already know the name of the message map function you need, so type the name into your class declaration, place the blinking text cursor on the name, and press F1. (For this to work, you must have Visual C++'s online documentation accessible on your hard disk or CD-ROM.) Developer Studio displays the help topic for the selected function. Just highlight the function's prototype in the help window, copy it to the Clipboard, and then paste it into your class's declaration.

The `OnLButtonDown()` prototype looks like this:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

The `afx_msg` part of the prototype does nothing more than mark the function as a message map function. You could leave the `afx_msg` off, and the function would still compile and execute just fine. However, Microsoft programming conventions suggest that you use `afx_msg` to distinguish message map functions from other types of member functions in your class.

Now, as long as you still have the function's prototype in the Clipboard, you might as well use it to create the function's definition in your class's implementation file. In the function's definition, you write the code that you want executed whenever the object receives the associated Windows message. In the case of your slowly growing MFC application, the OnLButtonDown() message response function simply displays a message box, indicating that the message was received and responded to by the frame window.

## Exploring the MFC App2 Application

If you're a little confused, take a look at Listings 4.6 and 4.7 for the CMainFrame class of the MFC App2 application, which adds message mapping to the MFC App1 application. These listings show the message maps in the context of the source code.



The complete source code and executable file for the MFC App2 application can be found in the CHAP04\MFCAPP2 directory of this book's CD-ROM.

### Listing 4.6 MAINFRM.H—The Header File for MFC App2's Frame-Window Class

---

```

////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//             represents the application's main window.
////////////////////////////////////
class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();
    ~CMainFrame();

    // Message map functions.
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);

    DECLARE_MESSAGE_MAP()
};

```

---

### Listing 4.7 MAINFRM.CPP—The Implementation File of MFC App2's Frame-Window Class

---

```

////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//             class, which represents the application's
//             main window.
////////////////////////////////////

#include <afxwin.h>
#include "mainfrm.h"

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_LBUTTONDOWN()

```

```
END_MESSAGE_MAP( )
```

```
////////////////////////////////////  
// CMainFrame: Construction and destruction.  
////////////////////////////////////  
CMainFrame::CMainFrame()  
{  
    Create(NULL, "MFC App2");  
}  
  
CMainFrame::~~CMainFrame()  
{  
}  
  
////////////////////////////////////  
// Message map functions.  
////////////////////////////////////  
void CMainFrame::OnLButtonDown(UINT nFlags, CPoint point)  
{  
    MessageBox("Got the click!", "MFC App2");  
}
```

---

When you compile and run the MFC App2 program, you see the application's main window. The application doesn't look much different from MFC App1, does it? Click inside the window, however, and a message box pops up (see Figure 4.4) to let you know that your message map is working.



**FIG. 4.4** MFC App2 uses message mapping to respond to mouse clicks.

---

**Note:**

Notice that the call to `MessageBox()` in the `OnLButtonDown()` function looks very different from the `MessageBox()` call that you might be used to seeing in conventional Windows programs. The `MessageBox()` function being called in `OnLButtonDown()` is a member function of the `CMainFrame` class (inherited from `CWnd`), rather than part of the Windows API. MFC's `MessageBox()` function requires no window handle (after all, the window class knows its own handle) and can get by with nothing but the text that should appear in the message box. Another MFC advantage!

---

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 5

# Painting a Window

- How device contexts control your displays
- How to respond to window-painting requests
- How to create and use fonts, pens, and brushes
- How to override class member functions

You now know how to create a simple MFC application without resorting to AppWizard. You managed to get a window up on the screen and even got the window to respond to mouse clicks, thanks to MFC's message maps. In this chapter, you extend your knowledge of MFC by seeing how to use device contexts and GDI objects in order to create your window's display.

## Understanding Device Contexts

As you know, every Windows application (in fact, every computer application) must manipulate data in some way. Most applications must also display data. Unfortunately, though, because of Windows' device independence, this task is not as straightforward in Windows as it is in DOS.

Although device independence forces you, the programmer, to deal with data displays indirectly, it helps you by ensuring that your programs run on all popular devices. In most cases, Windows handles devices for you through the device drivers that the user has installed on the system. These device drivers intercept the data the application needs to display and translates the data appropriately for the device on which it will appear, whether that is a screen, a printer, or some other output device.

To understand how all of this device independence works, imagine an art teacher trying to design a course of study appropriate for all types of artists. The teacher creates a course outline that stipulates the subject of a project, the suggested colors to be used, the dimensions of the finished project, and so on. What the teacher doesn't stipulate is the surface on which the project will be painted or the materials needed to paint on that surface. In other words, the teacher stipulates only general characteristics. The details of how these characteristics are applied to the finished project are left up to each artist.

The instructor in the preceding scenario is much like a Windows programmer. The programmer has no idea who might eventually use the program and what kind of system that user might have. The programmer can recommend the colors in which data should be displayed and the coordinates at which the data should appear, for example, but it is



the device driver—the Windows artist—that ultimately decides how the data appears.

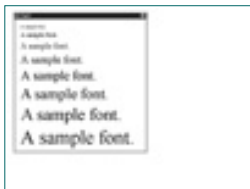
A system with a VGA monitor might display data with fewer colors than a system with a Super VGA monitor. Likewise, a system with a monochrome monitor displays the data in only a single color. Monitors with high resolutions can display more data than lower-resolution monitors. The device drivers, much like the artists in the imaginary art school, must take the display requirements and fine-tune them to the device on which the data will actually appear. It is a data structure called a *device context* that links the application to the device's driver.

A *device context* (DC) is little more than a data structure that keeps track of the attributes of a window's drawing surface. These attributes include the currently selected pen, brush, and font that will be used to draw on the screen. Unlike an artist, who can have many brushes and pens with which to work, a DC can use only a single pen, brush, or font at a time. If you want to use a pen that draws wider lines, for example, you need to create the new pen and then replace the DC's old pen with the new one. Similarly, if you want to fill shapes with a red brush, you must create the brush and "select it into the DC," which is how Windows programmers describe replacing a tool in a DC.

A window's client area is a versatile surface that can display anything a Windows program can draw. The client area can display any type of data because everything displayed in a window, whether it be text, spreadsheet data, a bitmap, or any other type of data, is displayed graphically. MFC helps you display data by encapsulating Windows' GDI functions and objects into its DC classes.

## Introducing the Paint1 Application

The sample program in this chapter shows you how to use MFC to display many types of data in an application's window. When you run the program, the main window appears, showing text drawn in various sized fonts (see Figure 5.1).



**FIG. 5.1** You can draw text using various sized fonts.

When you left-click the application's window, it switches the type of information it displays. For example, the first time you click, the window shows a series of blue lines that get thicker as they get closer to the window's bottom (see Figure 5.2). This screen is produced by creating new pens and drawing the lines with those pens.



**FIG. 5.2** An application can also display many different types of lines.

A second click brings up a display consisting of rectangles, each of which contains a different pattern (see Figure 5.3). The fill patterns in the rectangles are produced by the brushes created in the program. Finally, a third click brings you back to the font display.



**FIG. 5.3** Different brushes enable your window to display colors and patterns.

Listing 5.1 through Listing 5.4 are the source codes for the Paint1 application. Take some time now to run the application, as well as to look over the source code in the listings. In the sections that follow, you'll learn how the program produces its displays and how MFC simplifies the drawing of a window's displays.

---

**Note:**

The complete source code and executable file for the Paint1 application can be found in the CHAP05\PAINT1 directory of the CD-ROM included with this book.

---



### **Listing 5.1 PAINT1.H—The Header File of the Application Class**

---

```
////////////////////////////////////  
// PAINT1.H: Header file for the CPaintApp1 class, which  
//           represents the application object.  
////////////////////////////////////  
  
class CPaintApp1 : public CWinApp  
{  
public:  
    CPaintApp1();  
  
    // Virtual function overrides.  
    BOOL InitInstance();  
};
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
 • [Advanced Search](#)  
 • [Search Tips](#)

 **BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Exploring the Paint1 Application

If you took the time to look over the listings that constitute the Paint1 application, you're either now smugly confident that you know all there is to know about displaying data in a window or you feel like you just tried to read the latest John Grisham novel in Latin. Whichever category you fall into, you'll almost certainly want to read on. You smug folks might get some surprises, whereas the rest of you will discover that the code isn't anywhere near as complex as it might appear at first glance.

### Painting in an MFC Program

In Chapter 4, "Constructing an MFC Program from Scratch," you learned about message maps and how you can tell MFC which functions to call when it receives messages from Windows. One important message that every Windows program with a window must handle is WM\_PAINT. Windows sends the WM\_PAINT message to an application's window when the window needs to be redrawn. There are several events that cause Windows to send a WM\_PAINT message. The first event is simply the running of the program by the user. In a properly written Windows application, the application's window gets a WM\_PAINT message almost immediately after being run to ensure that the appropriate data is displayed from the very start.

Another time that a window might receive the WM\_PAINT message is when the window has been resized or has recently been uncovered—either fully or partially—by another window. In either case, part of the window that wasn't visible before is now on the screen and must be updated.

Finally, a program can indirectly send itself a WM\_PAINT message by invalidating its client area. Having this capability ensures that an application can change its window's contents almost any time it wants. For example, a word processor might invalidate its window after the user pastes some text from the Clipboard.

When you studied message maps, you learned to convert a message name to a message map macro and function name. You now know, for example, that the message map macro for a WM\_PAINT message is ON\_WM\_PAINT(). You also know that the matching message map function should be called OnPaint(). This is another case when MFC has already done most of the work of matching a Windows message with its message response function.

- See "Constructing an MFC Program from Scratch," p. 61

So, in order to paint your window's display, you need to add an `ON_WM_PAINT()` entry to your message map and then write an `OnPaint()` function. In the `OnPaint()` function, you write the code that will produce your window's display. Then, whenever Windows sends your application the `WM_PAINT` message, MFC will automatically call `OnPaint()`, which will draw the window's display just as you want it.

If you look near the top of Listing 5.4, which is the implementation file of the frame window class, you'll see the application's main message map, as shown in Listing 5.5.

---

**Listing 5.5 LST05\_05.CPP—The Message Map of the Paint1 Application**

---

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

---

As you can tell by the message map's entries, the application can respond to `WM_PAINT` and `WM_LBUTTONDOWN` messages. The `ON_WM_PAINT()` entry maps to the `OnPaint()` message map function. In the first line of that function, the program creates a DC for the client area of the frame window:

```
CPaintDC* paintDC = new CPaintDC(this);
```

`CPaintDC` is a special class for managing paint DCs, which are device contexts that are used only when responding to `WM_PAINT` messages. In fact, if you're going to use MFC to create your `OnPaint()` function's paint DC, you *must* use the `CPaintDC` class. This is because an object of the `CPaintDC` class does more than just create a DC; it also calls the `BeginPaint()` Windows API function in the class's constructor and calls `EndPaint()` in its destructor. When responding to `WM_PAINT` messages, calls to `BeginPaint()` and `EndPaint()` are required. The `CPaintDC` class handles this requirement without your having to get involved in all the messy details.

As you can see, the `CPaintDC` constructor takes a single argument, which is a pointer to the window for which you're creating the DC. Because the preceding code line uses the `new` operator to create the `CPaintDC` object dynamically on the heap, the program must call `delete` on the returned `CPaintDC` pointer when the program is finished with the DC. Otherwise, you not only leave unused objects floating around the computer's memory, but you also fail to call the `CPaintDC` object's constructor, which in turn never gets a chance to call `EndPaint()`.

After creating the paint DC, the `OnPaint()` function in Listing 5.4 uses its `m_display` data member to determine what type of display to draw in the window. The `m_display` data member can be equal to `Fonts`, `Pens`, or `Brushes`, three values that are defined as an enumeration in the `MAINFRM.H` file, like this:

```
enum {Fonts, Pens, Brushes};
```

The function checks `m_display` in a switch statement, calling `ShowFonts()`, `ShowPens()`, or `ShowBrushes()` as appropriate. In these three functions the `Paint1` application actually creates its displays. You'll examine these functions of your `CMainFrame` class in the sections to come. Notice that the pointer to the paint DC is passed as a parameter to `ShowFonts()`, `ShowPens()`, and `ShowBrushes()`, which must use the DC in order to draw in the application's window.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 5.1 LOGFONT Fields and Their Descriptions

Field	Description
lfHeight	Height of the font in logical units
lfWidth	Width of the font in logical units
lfEscapement	Angle at which to draw the text
lfOrientation	Character tilt in tenths of a degree
lfWeight	Field used to select normal (400) or boldface (700) text
lfItalic	A non-zero value indicates italics
lfUnderline	A non-zero value indicates an underlined font
lfStrikeOut	A non-zero value indicates a strikethrough font
lfCharSet	Font character set
lfOutPrecision	How to match the requested font to the actual font
lfClipPrecision	How to clip characters that run over the clip area
lfQuality	Print quality of the font
lfPitchAndFamily	Pitch and font family
lfFaceName	Typeface name

The LOGFONT structure holds a complete description of the font. This structure contains fourteen fields, although many of the fields can be set to 0 or the default values, depending on the program’s needs. In the ShowFonts() function, the Paint1 application creates its LOGFONT, as shown in Listing 5.7.

Listing 5.7 LST05\_07.CPP—Initializing a LOGFONT Structure

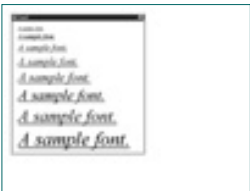
```

    LOGFONT logFont;
    logFont.lfHeight = 8;
    logFont.lfWidth = 0;
    logFont.lfEscapement = 0;
    logFont.lfOrientation = 0;
    logFont.lfWeight = FW_NORMAL;
    logFont.lfItalic = 0;
    logFont.lfUnderline = 0;
    logFont.lfStrikeOut = 0;
    logFont.lfCharSet = ANSI_CHARSET;
    logFont.lfOutPrecision = OUT_DEFAULT_PRECIS;
    logFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    logFont.lfQuality = PROOF_QUALITY;
    logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
    strcpy(logFont.lfFaceName, "Times New Roman");
  
```

In the preceding lines, the font is set to be eight pixels high, as determined by the value of the lfHeight

field. Note that, in most cases, you should set the width to 0, as determined by `lfWidth`, which enables Windows to select a width that best matches the height. You can, however, create compressed or expanded fonts by experimenting with the `lfWidth` field.

The font's italic, underline, and strikeout attributes can be turned on by supplying a non-zero value for the `lfItalic`, `lfUnderline`, and `lfStrikeOut` fields. For example, Figure 5.4 shows how the Paint1 application's font display would look if you set the `lfItalic` and `lfUnderline` members of the `LOGFONT` structure to 1.



**FIG. 5.4** You can create all types of fonts by manipulating the values in the `LOGFONT` structure.

In order to show the many fonts that are displayed in its window, the Paint1 application creates its fonts in a for loop, modifying the value of the `LOGFONT` structure's `lfHeight` member each time through the loop, using the loop variable, `x`, to calculate the new font height, like this:

```
logFont.lfHeight = 16 + (x * 8);
```

Because `x` starts at zero, the first font created in the loop will be sixteen pixels high. Each time through the loop, the new font will be eight pixels higher than the previous one.

[an error occurred while processing this directive]

The last line in `OnPaint()` deletes the `CPaintDC` object, `paintDC`, freeing it from memory, as well as ensuring that its destructor gets called properly. Note that, in most programs, you might find it more convenient to create your paint DC as a local variable on the stack, as is shown in Listing 5.6. When you do this, you no longer have to be concerned with deleting the object. It's automatically deleted when it goes out of scope. In the case of the Paint1 application, it's more efficient to pass a `CPaintDC` pointer as an argument to other functions than it is to pass the actual object.

**Listing 5.6 LST05\_06.CPP—Creating the `PaintDC` on the Stack**

```
void CMainFrame::OnPaint()  
{  
    CPaintDC paintDC(this);  
  
    // Drawing code goes here.
```

}

---

## Using Fonts

Fonts are one of the trickier GDI objects to handle, so you might as well get them out of the way first. In order to select and use fonts, you must be familiar with the LOGFONT structure, which contains a wealth of information about a font, and you must know how to create new fonts when they're needed. Moreover, there are more typefaces and font types than galaxies in the universe (okay, maybe not quite that many), which means you can never be sure exactly how your user's system is set up.

As I said, a Windows font is described in the LOGFONT structure, which is outlined in Table 5.1. The LOGFONT description in Table 5.1, however, gives only an overview of the structure. Before experimenting with custom fonts, you might want to look up this structure in your Visual C++ online help, where you'll find a more complete description of each of its fields, including the many constants that are already defined for use with the structure.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

After setting the font's height, the program creates a CFont object:

```
CFont* font = new CFont();
```

In case it's not obvious, CFont is MFC's font class. Using the CFont class, you can create and manipulate fonts using the class's member functions. One of the most important member functions is CreateFontIndirect(), which DisplayFonts() calls like this:

```
font->CreateFontIndirect(&logFont);
```

CreateFontIndirect() takes a single argument, which is the address of the LOGFONT structure that contains the font's attributes. When Windows receives the information stored in the LOGFONT structure, it will do its best to create the requested font. The font created isn't always exactly the font requested, so Windows fills in the LOGFONT structure with a description of the font it managed to create.

After calling CreateFontIndirect(), the CFont object has been associated with a Windows font. At this point, you can select the font into the DC, like this:

```
CFont* oldFont = paintDC->SelectObject(font);
```

Remember that in order to use a new graphical object with a DC, you must first select that object into the DC. The above call to the paint DC's SelectObject() member function replaces the current font in the DC with the new one. SelectObject()'s single parameter is the address of the new font. SelectObject() returns a pointer to the old font object that was deselected from the DC. You'll soon see why you must save this pointer.

After selecting the new font into the DC, you can use the font to draw text on the screen. In the ShowFonts() function, the first step in displaying text is to determine where in the window to draw the text:

```
position += logFont.lfHeight;
```

The local variable position holds the vertical position in the window at which the next line of text should be printed. This position depends upon the height of the current font. After all, if there's not enough space between the lines, the larger fonts will overlap the smaller ones. When Windows created the new font, it stored the font's height (which is most likely the height you requested, but, then again, maybe not) in the LOGFONT structure's lfHeight member. By adding the

value stored in `lfHeight`, the program can determine the next position at which to display the line of text, using the DC object's `TextOut()` member function:

```
paintDC->TextOut(20, position, "A sample font.");
```

Here, `TextOut()`'s three arguments are the X,Y coordinates in the window at which to print the text and the text to print. `TextOut()` actually has a fourth argument, which is the number of characters to print. If you leave this last parameter off, MFC just assumes you want to display the entire string given as the second argument.

Now you get to see why the program saved the pointer of the old font that was deselected from the DC when the program selected the new font. You must never delete a GDI object, such as a font, from a DC while it's still selected into the DC. That means you must first deselect the new font from the DC before deleting it in preparation for creating the next font. Unfortunately, if you search through your Windows programming manuals, you'll discover that there is no `DeselectObject()` function. This actually makes sense when you think about it. If you were able to deselect GDI objects without selecting new ones, you could leave the DC without a pen, brush, or other important object. So, the only way to deselect an object is to select a new object into the DC. Therefore, to deselect the new font, the `Paint1` application selects the old font back into the DC, like this:

```
paintDC->SelectObject(oldFont);
```

This time the program doesn't bother to save the pointer returned from `SelectObject()` because that pointer is for the new font that the program created. The program already has that pointer stored in `font`, so a quick call to `delete` gets rid of the font object. Consequently, the program can create the next font it needs for the display:

```
delete font;
```

## Using Pens

You'll be pleased to know that pens are much easier to deal with than fonts, mostly because you don't have to fool around with complicated data structures like `LOGFONT`. In fact, to create a pen, you need only supply the pen's line style, thickness, and color. In its window, the `Paint1` application's `ShowPens()` function displays lines drawn using different pens created within a for loop. Within the loop, the program first creates a custom pen, like this:

```
CPen* pen = new CPen(PS_SOLID, x*2+1, RGB(0, 0, 255));
```

The first argument just shown is the line's style, which can be one of the styles listed in Table 5.2. Note that only solid lines can be drawn with different thicknesses. Patterned lines always have a thickness of 1. The second argument above is the line thickness, which, in the `ShowPens()` function, is calculated using the loop variable `x` as a multiplier.

Finally, the third argument is the line's color. The RGB macro takes three values for the red, green, and blue color components and converts them into a valid Windows color reference. The values for the red, green, and blue color components can be anything from zero to 255—the higher the value, the brighter the color component. The preceding line creates a bright blue pen. If all the color values were zero, the pen would be black, whereas if the color values were all 255, the pen would be white.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 5.2 Pen Styles

Style	Meaning
PS_DASH	Specifies a pen that draws dashed lines
PS_DASHDOT	Specifies a pen that draws dash-dot patterned lines
PS_DASHDOTDOT	Specifies a pen that draws dash-dot-dot patterned lines
PS_DOT	Specifies a pen that draws dotted lines
PS_INSIDEFRAME	Specifies a pen that’s used with shapes, where the line’s thickness must not extend outside of the shape’s frame
PS_NULL	Specifies a pen that draws invisible lines
PS_SOLID	Specifies a pen that draws solid lines

After creating the new pen, the program selects it into the DC, saving the pointer to the old pen, like this:

```
CPen* oldPen = paintDC->SelectObject(pen);
```

After the pen is selected into the DC, the program can draw a line with the pen. To do this, the program first calculates a vertical position for the new line and then calls the paint DC’s MoveTo() and LineTo() member functions, like this:

```
position += x * 2 + 10;
paintDC->MoveTo(20, position);
paintDC->LineTo(400, position);
```

The MoveTo() function positions the starting point of the line, whereas the LineTo() function draws a line—using the pen currently selected into the DC—from the point set with MoveTo() to the coordinates given as the function’s two arguments.

Finally, the last step is to restore the DC by reselecting the old pen and deleting the new pen, which is no longer selected into the DC:

```
paintDC->SelectObject(oldPen);
delete pen;
```

Tip:

If you want to control the style of a line’s end points or want to create your own custom patterns for pens, you can use the alternate CPen constructor, which requires a few more arguments than the CPen constructor described in this section. To learn how to use this alternate constructor, look up CPen in your Visual C++ online documentation.

Using Brushes

Creating and using brushes in an MFC program is not unlike using pens. In fact, just as with pens, you can create both solid and patterned brushes. You can even create brushes from bitmaps that contain your own custom fill patterns. As you've seen, the Paint1 application displays rectangles that have been filled by both patterned and solid rectangles. These rectangles are produced in the ShowBrushes() function, which, like the font and pen functions you've already examined, creates its graphical objects within a for loop. In the first line of the loop's body, the program defines a pointer to a CBrush object:

```
CBrush* brush;
```

Then, depending on the value of the loop variable x, the program creates either a solid or a patterned brush, as shown in Listing 5.8.

---

**Listing 5.8 LST05\_08.CPP—Creating CBrush Objects**

---

```
        if (x == 6)
            brush = new CBrush( RGB(0,255,0) );
        else
            brush = new CBrush(x, RGB(0,160,0));
```

---

In Listing 5.8, if x equals 6, the program calls the version of the CBrush constructor that creates a solid pen. The constructor's single argument is a COLORREF value, which is easily produced using the RGB macro to which you were introduced in the section "Using Pens." For any other value of x, the program creates a patterned brush, using x as the pattern index. The second CBrush constructor takes the pattern index and the brush color as its two arguments. Although not used in the preceding code segment, Windows defines several constants for the brush patterns (or hatch styles, as they're often called). Those constants are HS\_BDIAGONAL, HS\_CROSS, HS\_DIAGCROSS, HS\_FDIAGONAL, HS\_HORIZONTAL, and HS\_VERTICAL.

After the program has created the new brush, a call to the paint DC's SelectObject() member function selects the brush into the DC and returns a pointer to the old brush:

```
CBrush* oldBrush = paintDC->SelectObject(brush);
```

Now the program calculates a new drawing position, and then draws a rectangle with the new brush:

```
        position += 50;
        paintDC->Rectangle(20, position, 400, position + 40);
```

Rectangle() is just one of the shape-drawing functions you can call. Rectangle() takes as arguments the coordinates of the rectangle's upper-left and lower-right corners. When you run the Paint1 application and look at the brush window, you'll see that each rectangle is bordered by a thin black line. This line was drawn by the DC's default black pen. If you had selected a different pen into the DC, Windows would have used that pen to draw the rectangle's border. For example, in Figure 5.5, the Paint1 program shows rectangles drawn with a six-pixel thick pen.



**FIG. 5.5** Shape-drawing functions frequently draw borders with the currently selected pen.

After drawing a rectangle, the program deselects the new brush from the DC and deletes it:

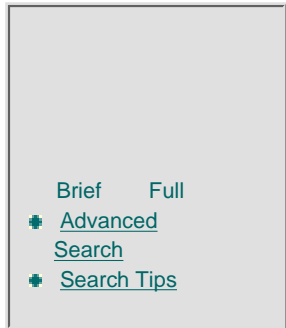
```
paintDC->SelectObject(oldBrush);  
delete brush;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced](#)  
[Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Switching the Display

As you know, when you click in the Paint1 application's window, the window's display changes. This seemingly magical feat is actually easy to accomplish. The program routes WM\_LBUTTONDOWN messages to the OnLButtonDown() message response function, just as the sample program in Chapter 4, "Constructing an MFC Program from Scratch," did. But rather than display a message box, this version of OnLButtonDown() is charged with keeping the window updated with the selected display. At the program start-up, the CMainFrame class's constructor initializes its data member m\_display to Fonts so that the window initially appears with the fonts displayed. When the user clicks in the window, the OnLButtonDown() function changes the value of m\_display, as shown in Listing 5.9.

### Listing 5.9 LST05\_09.CPP—Changing the Value of m\_display

```

if (m_display == Fonts)
    m_display = Pens;
else if (m_display == Pens)
    m_display = Brushes;
else
    m_display = Fonts;
    
```

As you can see, depending on its current value, m\_display gets set to the next display type in the series. Of course, just changing the value of m\_display doesn't accomplish much. The program still needs to redraw the contents of its window. Because OnPaint() determines which display to paint based on the value of m\_display, all the program needs to do is to get OnPaint() to execute. This task is accomplished by calling the CMainFrame class's Invalidate() function:

```
Invalidate();
```

A call to Invalidate() tells Windows that all of the window needs to be repainted. This causes Windows to generate a WM\_PAINT message for the window. Thanks to MFC's message mapping, the WM\_PAINT message gets routed to OnPaint(). Although it's not used in the preceding example, Invalidate() actually has one argument, which MFC gives the default value of TRUE. This Boolean argument tells Windows whether to erase the window's background. If you use FALSE for this argument, Windows leaves the background alone. In Figure 5.6,

you can see what happens to the Paint1 application if Invalidate() gets called with an argument of FALSE.



**FIG. 5.6** Without erasing the background, the Paint1 application's window gets a bit messy.

## Sizing and Positioning the Window

In Chapter 4, you saw how to initially size and position a window when calling the window's Create() member function. You can also size and position a window by overriding the PreCreateWindow() function of the window class. This method of positioning a window is especially useful in an AppWizard-generated program because you don't usually call Create() directly in such applications. Although the Paint1 application wasn't created by AppWizard, it does override its window class's PreCreateWindow() function to position the window, as shown in Listing 5.10.

**Listing 5.10 LST05\_10.CPP—Overriding PreCreateWindow()**

---

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Set size of the main window.
    cs.cx = 440;
    cs.cy = 460;

    // Call the base class's version.
    BOOL returnCode = CFrameWnd::PreCreateWindow(cs);

    return returnCode;
}
```

---

The PreCreateWindow() function, which MFC calls right before the window element that'll be associated with the class is created, receives one parameter: a reference to a CREATESTRUCT structure. The CREATESTRUCT structure contains essential information about the window that's about to be created and is declared by Windows, as shown in Listing 5.11.

**Listing 5.11 LST05\_11.CPP—The CREATESTRUCT Structure**

---

```
typedef struct tagCREATESTRUCT {
    LPVOID    lpCreateParams;
    HANDLE    hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
```



```
int      cy;  
int      cx;  
int      y;  
int      x;  
LONG     style;  
LPCSTR   lpzName;  
LPCSTR   lpzClass;  
DWORD    dwExStyle;  
} CREATESTRUCT;
```

---

If you've programmed Windows without application frameworks, such as MFC, you'll recognize the information stored in the CREATESTRUCT structure. You supply much of this information when calling the Windows API function `CreateWindow()` to create your application's window. Of special interest to MFC programmers are the `cy`, `cx`, `y`, and `x` members of this structure. By changing `cy` and `cx`, you can set the width and height, respectively, of the window. Similarly, modifying `y` and `x` changes the window's position. By overriding `PreCreateWindow()`, you get a chance to fiddle with the CREATESTRUCT structure before Windows uses it to create the window.

It's important that after your own code in `PreCreateWindow()` you call the base class's `PreCreateWindow()`. Failure to do this will leave you without a valid window because MFC never gets a chance to pass the CREATESTRUCT structure on to Windows, so Windows never creates your window. When overriding class member functions, you often need to call the base class's version, either before or after your own code, depending on the function. You'll learn more about this as you work your way through this book. Also, the descriptions of member functions in your Visual C++ online documentation indicate whether or not the base class's version must be called.

You're really starting to master MFC now. Take some time at this point to look over the CDC class and the several classes derived from CDC in your Visual C++ online documentation. You'll discover a wealth of member functions you can use to create displays for your windows. Remember that `CPaintDC` is just one type of CDC-derived class, one that's used specifically in the `OnPaint()` message response function. If you want to create a DC to paint your window in some other part of your program, you'll probably want to use the `CClientDC` class.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 6

# Using Menus

- How to create menu resources
- Discover how message maps work
- Learn to write message response functions
- How to create command UI functions
- How to use variables to keep track of menu states

A computer application without a menu bar is like a restaurant that sends you to the kitchen to find your own meal. Fortunately, the days of hidden keyboard commands and tricky control codes are gone. Your application's users expect your program to display its commands proudly in plain view. Moreover, your users expect to find familiar commands in familiar places.

This consistency between applications is one thing that makes Windows applications easier to use. When users load a new application, they may not know exactly how to use it, but they at least know where to find the Open command (the File menu) or where to find the Cut, Copy, and Paste commands (the Edit menu). You can be sure that if users don't find these commands and others like them in the places they expect the commands to be, they may stop using your application.

However, handling menus with MFC is a lot different from handling them with a conventional Windows programming language, such as C, because MFC incorporates message maps to link menu commands with the functions that handle those commands. In addition, MFC's `CCommandUI` class makes it easier than falling on ice to keep your menu commands enabled or disabled, check marked, or bulleted. In this chapter, you get an in-depth look at how the MFC menus work.

## Understanding Menus

Although creating and manipulating MFC menus takes quite a bit of program code, the process is fairly simple. You need only to create the menu with the Visual C++ built-in menu editor, add message response functions to your window class, and add the appropriate entries to your message map. Then, after you load the menu in your program, the menu takes care of itself, and the appropriate message response functions are called whenever the user selects a

menu item. The steps for adding menu support to your MFC application are as follows:

1. Create your menu resource using the Visual C++ menu editor. This process defines the menu titles that will appear in the menu bar as well as the menu commands that'll be in each pop-up menu.
2. Add menu state variables and message map functions to your window class's declaration. The message map functions can be conventional message response functions or special functions that control the appearance of menu commands.
3. Add the appropriate entries (using the predefined macros) to your window class's message map table. There are four commonly used macros: `ON_COMMAND`, `ON_COMMAND_RANGE`, `ON_UPDATE_COMMAND_UI`, and `ON_UPDATE_COMMAND_UI_RANGE`.
4. Write the message respond and update-command-UI functions that you have listed in your message map. The message response functions respond to the user's commands, whereas the update-command-UI functions determine whether commands are enabled, checked, bulleted, and so on.

In the rest of this chapter, you'll see, not only how to perform the preceding steps, but also how to assign command IDs to your menus and how to ensure that those menu IDs are accessible to the rest of your program. In the next section, you learn to use the Visual C++ menu editor to create your menu resource.

## Creating a Menu Resource

As you now know, the first step toward adding a menu to your MFC application is creating the menu resource, which acts as a sort of template for Windows. When Windows sees the menu resource in your program, it uses the commands in the resource to construct the menu bar for you. To create a menu resource, just perform the following steps:

1. Select the Insert, Resource command from Developer Studio's menu bar. The Insert Resource dialog box appears, as shown in Figure 6.1.



**FIG. 6.1** The Insert Resource dialog box enables you to select the type of resource you want to add to your project.

2. Double-click Menu in the Resource Type box. The menu editor appears in one of Developer Studio's panes (see Figure 6.2).
3. Double-click the blank menu to define your first menu. The Menu Item Properties property sheet appears (see Figure 6.3).



**FIG. 6.2** You actually build your menu resource in the menu editor.



**FIG. 6.3** You define each menu title and command using the Menu Item Properties property sheet.

4. Type the menu's name into the Caption box, and press Enter. The menu editor then adds the new menu title to the menu bar.
5. Double-click the blank menu item to start defining the first menu command in the new menu you created in Step 4. Type the command's ID into the ID box, type the command's caption into the Caption box (see Figure 6.4), and then press Enter.



**FIG. 6.4** Unlike menu titles, which have only a caption, a menu command has both an ID and a caption.

6. Repeat the appropriate steps above to define all your menus and menu commands.

---

**Note:**

When you create menu captions, you can specify a hotkey by placing an ampersand (&) immediately before the letter in the caption that the user can press to select that command. Windows automatically underlines the hotkey when it displays the menu caption.

---

## Defining Menu IDs

So that Windows can tell the application which menu command the user has selected, you must define IDs for all commands in your menus. The ID is a numerical value represented by a constant. However, you don't have to worry about an ID's actual value because when you add a new command-ID constant to your menu resource, the menu editor automatically gives it the next consecutive ID value.

You can choose any constant names you want for IDs. However, most MFC programmers follow a simple convention: They start the ID with the prefix `ID_`, followed by the menu's title and command separated by an underscore. So, a command ID for an Options menu's Color command would be

ID\_OPTIONS\_COLOR.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 6.1 Predefined Command IDs and Their Values

Command ID	Value
ID_APP_ABOUT	0xE140
ID_APP_EXIT	0xE141
ID_EDIT_CLEAR	0xE120
ID_EDIT_CLEAR_ALL	0xE121
ID_EDIT_COPY	0xE122
ID_EDIT_CUT	0xE123
ID_EDIT_FIND	0xE124
ID_EDIT_PASTE	0xE125
ID_EDIT_PASTE_LINK	0xE126
ID_EDIT_PASTE_SPECIAL	0xE127
ID_EDIT_REDO	0xE12C
ID_EDIT_REPEAT	0xE128
ID_EDIT_REPLACE	0xE129
ID_EDIT_SELECT_ALL	0xE12A
ID_EDIT_UNDO	0xE12B
ID_FILE_CLOSE	0xE102
ID_FILE_NEW	0xE100
ID_FILE_OPEN	0xE101
ID_FILE_PAGE_SETUP	0xE105
ID_FILE_PRINT	0xE107
ID_FILE_PRINT_PREVIEW	0xE109
ID_FILE_PRINT_SETUP	0xE106
ID_FILE_SAVE	0xE103
ID_FILE_SAVE_AS	0xE104
ID_WINDOW_ARRANGE	0xE131
ID_WINDOW_CASCADE	0xE132
ID_WINDOW_NEW	0xE130
ID_WINDOW_TILE_HORZ	0xE133
ID_WINDOW_TILE_VERT	0xE134

**Note:**

When defining menu items that will use previously defined IDs, do not type the ID into the Menu Item property sheet. If you just type the menu command—for example, E&amp;xit—Developer Studio will attempt to match the command with a predefined ID. If you try to add the ID manually, you will get a compiler warning because the ID will be defined both in your RESOURCE.H file and in the predefined IDs.

Dealing with Resource Files

After creating your menu resource (or any other type of resource for that matter), Developer Studio creates at least two files that you need to add to your application. The first file is called RESOURCE.H.

This file contains all of the resource IDs (in this case, menu IDs) that you’ve defined. You must include RESOURCE.H in any file that refers to the constants you’ve defined. Otherwise, the compiler will have no idea what the constants represent and will complain with a string of error messages. Listing 6.1 shows the RESOURCE.H file that was created by Developer Studio for the Menu application you’ll examine later in this chapter.

**Listing 6.1 RESOURCE.H—The RESOURCE.H File Holds All the Constants You Defined in Your Resource File**

---

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by menu.rc
//

#define IDR_MENU1 101
#define IDM_CHECKS_OPTION1 40002
#define IDM_CHECKS_OPTION2 40003
#define IDM_CHECKS_OPTION3 40004
#define IDM_ENABLE_ENABLE 40005
#define IDM_ENABLE_OPTION1 40006
#define IDM_ENABLE_OPTION2 40007
#define IDM_BULLETS_OPTION1 40008
#define IDM_BULLETS_OPTION2 40009
#define IDM_BULLETS_OPTION3 40010
#define IDM_SWITCH_ONOFF 40011

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 104
#define _APS_NEXT_COMMAND_VALUE 40012
#define _APS_NEXT_CONTROL_VALUE 1000
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

---

The first block of constants in Listing 6.1 are the constants that were defined when creating the application’s menus. The second block of constants is used internally by Developer Studio to help manage the other symbols. For example, you can see that the last constant defined in the first block has a value of 40011, which means the next available value is 40012. The constant APS\_NEXT\_COMMAND\_VALUE is not so coincidentally given the value 40012.

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

In Figure 6.4, you can see an application's Exit command being defined. In this case, the command ID is previously defined by Visual C++, which defines an entire set of common command IDs for your application's File, Edit, View, Window, and Help menus. Table 6.1 lists the most commonly used of these command IDs along with their hexadecimal values.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Brief Full

- Advanced
- Search
- Search Tips



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The second file that Developer Studio creates will have an .RC extension. This is the resource script that defines all your applications resources, including not just menus, but also dialog boxes, string tables, icons, cursors, and more. The resource script is a lot like a source code file, except it's written in a language that the resource compiler understands. The resource compiler takes the .RC file and compiles it into a .RES file, which is the binary representation of your application's resources. Listing 6.2 shows the MENU.RC file, which is the resource script for the menus you'll experiment with when you examine the Menu application later in this chapter, in the section titled "Exploring the Menu Application."

**Listing 6.2 MENU.RC—The Resource Script of the Menu Application**

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

////////////////////////////////////
//
// Menu
//

IDR_MENU1 MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit", ID_APP_EXIT
    END
    POPUP "&Enable"
    BEGIN
        MENUITEM "&Enable", IDM_ENABLE_ENABLE
        MENUITEM SEPARATOR
        MENUITEM "Option&1", IDM_ENABLE_OPTION1
        MENUITEM "Option&2", IDM_ENABLE_OPTION2
    END
    POPUP "&Checks"
```

```

        BEGIN
            MENUITEM "Option&1",          IDM_CHECKS_OPTION1
            MENUITEM "Option&2",          IDM_CHECKS_OPTION2
            MENUITEM "Option&3",          IDM_CHECKS_OPTION3
        END
    POPUP "&Bullets"
    BEGIN
        MENUITEM "Option&1",          IDM_BULLETS_OPTION1
        MENUITEM "Option&2",          IDM_BULLETS_OPTION2
        MENUITEM "Option&3",          IDM_BULLETS_OPTION3
    END
    POPUP "&Switch"
    BEGIN
        MENUITEM "&On",                IDM_SWITCH_ONOFF
    END
END

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif      // APSTUDIO_INVOKED

#endif      // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif      // not APSTUDIO_INVOKED

```

---

The specific area of interest in Listing 6.2 is the section of resource script following the Menu comment block. There you can see the actual commands that the resource compiler uses to create the binary version of the application's menus. Many programmers still write their resource scripts by hand, and most like to tinker with the script produced by programs, such as Developer Studio. If you're interested in learning more about resource scripts (always a good idea), consult your favorite Windows programming book.

---

**Note:**

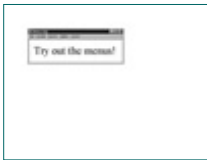
When you've had a chance to explore the Menu application that's presented later in this chapter (in the section titled "Exploring the Menu Application"), you might want to compare the resource script to the application's actual menus. You'll immediately see how the script's structure is converted into the menu hierarchy that you found in the application's menu bar.

---

Whether you want to experiment with the .RC file or just use it as is (the best choice unless you understand resource scripts thoroughly), you need to make it part of your project before Developer Studio can compile it and add it to your application's executable file. To add the resource script to your project, choose Insert, Files Into Project from Developer Studio's menu bar. When you do, the Insert Files into Project dialog box appears. Find the .RC file, and double-click it. Developer Studio adds the file to your project. The next time you compile your project, Developer Studio will automatically invoke the resource compiler to compile the script into a binary .RES file.

## Introducing the Menu Application

On this book's CD-ROM (or, if you installed the CD-ROM, on your hard disk) in the CHAP06 folder, you'll find the Menu sample program. To run the program, double-click the MENU.EXE file. When you do, you see the window shown in Figure 6.5. Besides the silly message in the window, this application sports a full set of menus that do all kinds of fancy tricks.



**FIG. 6.5** The menu application boasts a full menu bar.

First, turn your attention to the Enable menu. Click the menu's title, and you'll see that the menu has a command named Enable as well as two disabled menu commands called Option1 and Option2 (see Figure 6.6). Currently, Option1 and Option2 are disabled, meaning that they are grayed out and you can't select them.



**FIG. 6.6** At first, two commands on the Enable menu are disabled.

To enable these two commands, you must first select the Enable command. When you do, a check mark appears next to the Enable command, and the Option1 and Option2 commands are enabled, as shown in Figure 6.7. If you click one of the newly enabled commands, a message box appears confirming that the command was received by the application.



**FIG. 6.7** The Enable command enables the remaining commands in the menu.

[Previous](#) [Table of Contents](#) [Next](#)

Brief
 Full

+ [Advanced Search](#)

+ [Search Tips](#)



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

Next, click the Checks menu to display its commands, which are named Option1, Option2, and Option3. (Man, I’m really creative with those command names, eh?) Figure 6.8 shows what the menu looks like. When you click a command in the menu, a check mark appears next to the command. You can click any or all of the commands in the Check menu (see Figure 6.9).



**FIG. 6.8** The Checks menu commands start off with no check marks.



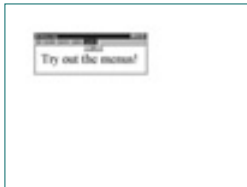
**FIG. 6.9** You can check mark any of the commands on the Checks menu by clicking the command.

The Bullets menu hides a few cool tricks, too. When you look at the menu, the first command (yes, it’s called Option1) is marked with a bullet, which indicates that that option is currently selected. As you’ll soon discover, only one of the three options can be selected at a time. Whichever option you click inherits the bullet (see Figure 6.10).



**FIG. 6.10** The options in the Bullets menu work like radio buttons.

Finally, the Switch menu demonstrates how a program can change the command captions in a menu. When you first open the Switch menu, its single command is captioned On. When you click the command, it changes to Off (see Figure 6.11). Click again, and it’s back to On.



**FIG. 6.11** The Switch menu’s single command toggles between On and Off.

## Exploring the Menu Application

Now that you've had a chance to use the Menu application, you're probably dying to see how it all works. Listings 6.3 and 6.4 are the source code for the Menu application's CMainFrame class. (There's nothing new or interesting about Menu's application class, so it's not shown here.) Listing 6.3 is the header file where the class is declared, whereas Listing 6.4 is the class's implementation file where its many functions are defined.

### Listing 6.3 MAINFRM.H—The Header File of the Frame Window Class

---

```
////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//             represents the application's main window.
////////////////////////////////////

enum {Option1, Option2, Option3};

class CMainFrame : public CFrameWnd
{
// Protected data members.
protected:
    BOOL m_enabled;
    BOOL m_checkOption1;
    BOOL m_checkOption2;
    BOOL m_checkOption3;
    UINT m_bulletOption;
    BOOL m_switch;

// Constructor and destructor.
public:
    CMainFrame();
    ~CMainFrame();

// Overrides.
protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Message map functions.
protected:
    // System message handlers.
    afx_msg void OnPaint();

    // Menu command message handlers.
    afx_msg void OnEnableEnable();
    afx_msg void OnEnableOption1();
    afx_msg void OnEnableOption2();
    afx_msg void OnChecksOption1();
    afx_msg void OnChecksOption2();
    afx_msg void OnChecksOption3();
    afx_msg void OnBullets(UINT nID);
    afx_msg void OnSwitchOnOff();
}
```

```

        // Update command UI handlers.
afx_msg void OnUpdateEnableEnableUI(CCmdUI* pCmdUI);
afx_msg void OnUpdateEnableOption1UI(CCmdUI* pCmdUI);
afx_msg void OnUpdateEnableOption2UI(CCmdUI* pCmdUI);
afx_msg void OnUpdateChecksOption1UI(CCmdUI* pCmdUI);
afx_msg void OnUpdateChecksOption2UI(CCmdUI* pCmdUI);
afx_msg void OnUpdateChecksOption3UI(CCmdUI* pCmdUI);
afx_msg void OnUpdateBulletsUI(CCmdUI* pCmdUI);
afx_msg void OnUpdateSwitchOnOffUI(CCmdUI* pCmdUI);

// Protected member functions.
protected:
    void ShowMessage(CPaintDC* paintDC);

    DECLARE_MESSAGE_MAP()
};

```

---

#### **Listing 6.4 MAINFRM.CPP—The Implementation File of the Frame Window Class**

---

```

////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//               class, which represents the application's
//               main window.
////////////////////////////////////

#include <afxwin.h>
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    // Message map entries for system messages.
    ON_WM_PAINT()

    // Message map entries for menu commands.
    ON_COMMAND(IDM_ENABLE_ENABLE, OnEnableEnable)
    ON_COMMAND(IDM_ENABLE_OPTION1, OnEnableOption1)
    ON_COMMAND(IDM_ENABLE_OPTION2, OnEnableOption2)
    ON_COMMAND(IDM_CHECKS_OPTION1, OnChecksOption1)
    ON_COMMAND(IDM_CHECKS_OPTION2, OnChecksOption2)
    ON_COMMAND(IDM_CHECKS_OPTION3, OnChecksOption3)
    ON_COMMAND(IDM_SWITCH_ONOFF, OnSwitchOnOff)

    ON_COMMAND_RANGE(IDM_BULLETS_OPTION1,
        IDM_BULLETS_OPTION3, OnBullets)

    // Message map entries for update command UI handlers.
    ON_UPDATE_COMMAND_UI(IDM_ENABLE_ENABLE,
        OnUpdateEnableEnableUI)
    ON_UPDATE_COMMAND_UI(IDM_ENABLE_OPTION1,

```

```

        OnUpdateEnableOption1UI)
ON_UPDATE_COMMAND_UI(IDM_ENABLE_OPTION2,
    OnUpdateEnableOption2UI)
ON_UPDATE_COMMAND_UI(IDM_CHECKS_OPTION1,
    OnUpdateChecksOption1UI)
ON_UPDATE_COMMAND_UI(IDM_CHECKS_OPTION2,
    OnUpdateChecksOption2UI)
ON_UPDATE_COMMAND_UI(IDM_CHECKS_OPTION3,
    OnUpdateChecksOption3UI)
ON_UPDATE_COMMAND_UI(IDM_SWITCH_ONOFF,
    OnUpdateSwitchOnOffUI)

    ON_UPDATE_COMMAND_UI_RANGE(IDM_BULLETS_OPTION1,
        IDM_BULLETS_OPTION3, OnUpdateBulletsUI)
END_MESSAGE_MAP()

////////////////////////////////////
// CMainFrame: Construction and destruction.
////////////////////////////////////
CMainFrame::CMainFrame()
{
    // Create the main frame window.
    Create(NULL, "Menu App", WS_OVERLAPPEDWINDOW, rectDefault,
        NULL, MAKEINTRESOURCE(IDR_MENU1));

    // Initialize class data members.
    m_enabled = FALSE;
    m_checkOption1 = FALSE;
    m_checkOption2 = FALSE;
    m_checkOption3 = FALSE;
    m_bulletOption = Option1;
    m_switch = FALSE;
}

CMainFrame::~CMainFrame()
{
}

////////////////////////////////////
// Overrides.
////////////////////////////////////
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Set size of the main window.
    cs.cx = 380;
    cs.cy = 140;

    // Call the base class's version.
    BOOL returnCode = CFrameWnd::PreCreateWindow(cs);

    return returnCode;
}

```

```

}

////////////////////////////////////
// Message map functions.
////////////////////////////////////
void CMainFrame::OnPaint()
{
    CPaintDC* paintDC = new CPaintDC(this);
    ShowMessage(paintDC);
    delete paintDC;
}

void CMainFrame::OnEnableEnable()
{
    m_enabled = !m_enabled;
}

void CMainFrame::OnEnableOption1()
{
    MessageBox("Enable/Option1");
}

void CMainFrame::OnEnableOption2()
{
    MessageBox("Enable/Option2");
}

void CMainFrame::OnChecksOption1()
{
    m_checkOption1 = !m_checkOption1;
}

void CMainFrame::OnChecksOption2()
{
    m_checkOption2 = !m_checkOption2;
}

void CMainFrame::OnChecksOption3()
{
    m_checkOption3 = !m_checkOption3;
}

void CMainFrame::OnBullets(UINT nID)
{
    if (nID == IDM_BULLETS_OPTION1)
        m_bulletOption = Option1;
    else if (nID == IDM_BULLETS_OPTION2)
        m_bulletOption = Option2;
    else
        m_bulletOption = Option3;
}

```



```

void CMainFrame::OnSwitchOnOff()
{
    m_switch = !m_switch;
}

////////////////////////////////////
// Update Command UI Handlers
////////////////////////////////////
void CMainFrame::OnUpdateEnableEnableUI(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_enabled);
}

void CMainFrame::OnUpdateEnableOption1UI(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_enabled);
}

void CMainFrame::OnUpdateEnableOption2UI(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_enabled);
}

void CMainFrame::OnUpdateChecksOption1UI(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_checkOption1);
}

void CMainFrame::OnUpdateChecksOption2UI(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_checkOption2);
}

void CMainFrame::OnUpdateChecksOption3UI(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_checkOption3);
}

void CMainFrame::OnUpdateBulletsUI(CCmdUI* pCmdUI)
{
    if (pCmdUI->m_nID == IDM_BULLETS_OPTION1)
        pCmdUI->SetRadio(m_bulletOption == Option1);
    else if (pCmdUI->m_nID == IDM_BULLETS_OPTION2)
        pCmdUI->SetRadio(m_bulletOption == Option2);
    else
        pCmdUI->SetRadio(m_bulletOption == Option3);
}

void CMainFrame::OnUpdateSwitchOnOffUI(CCmdUI* pCmdUI)
{

```

```

        if (m_switch)
            pCmdUI->SetText( "&amp;Off" );
        else
            pCmdUI->SetText( "&amp;On" );
    }

    //////////////////////////////////////
    // Protected member functions.
    //////////////////////////////////////
void CMainFrame::ShowMessage(CPaintDC* paintDC)
{
    // Initialize a LOGFONT structure for the fonts.
    LOGFONT logFont;
    logFont.lfHeight = 48;
    logFont.lfWidth = 0;
    logFont.lfEscapement = 0;
    logFont.lfOrientation = 0;
    logFont.lfWeight = FW_NORMAL;
    logFont.lfItalic = 0;
    logFont.lfUnderline = 0;
    logFont.lfStrikeOut = 0;
    logFont.lfCharSet = ANSI_CHARSET;
    logFont.lfOutPrecision = OUT_DEFAULT_PRECIS;
    logFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    logFont.lfQuality = PROOF_QUALITY;
    logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
    strcpy(logFont.lfFaceName, "Times New Roman");

    // Create a new font and select it into the DC.
    CFont font;
    font.CreateFontIndirect(&logFont);
    CFont* oldFont = paintDC->SelectObject(&font);

    // Print text with the new font.
    paintDC->TextOut(20, 20, "Try out the menus!");

    // Restore the old font to the DC.
    paintDC->SelectObject(oldFont);
}

```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Declaring Menu State Variables

If you have very simple menus in your applications—menus that require no special handling, such as check marking or disabling—you probably don’t need menu state variables in your program. However, most full-fledged Windows applications have at least a few menu commands that require special handling. The most common example is a menu command that must be enabled or disabled depending upon the application’s current status. For example, the Edit, Paste command should not be enabled unless there’s something in the Clipboard to paste. Another example might be a menu of options that can be turned on or off. If an option is on, it should be check marked or bulleted.

How you use menu state variables in your application depends, of course, on how you need to manage the application’s menu commands. Usually, you’ll use Boolean values to determine the state of a menu command. A set of options, for example, will have an equivalent set of Boolean variables: one for each option. If the option is on, the equivalent variable is set to TRUE; if the option is off, the variable is set to FALSE. Having menu state variables not only helps your program know how to respond to the user’s commands, but also enables your application to easily maintain the menu commands’ appearances, as you’ll see later in this chapter, in the section titled “Writing Update-Command-UI Functions.”

If you look at the declaration of the CMainFrame class, you’ll see the data member declarations shown in Listing 6.5.

### Listing 6.5 LST06\_05.CPP—The Menu State Variables of the CMainFrame Class

```

BOOL m_enabled;
BOOL m_checkOption1;
BOOL m_checkOption2;
BOOL m_checkOption3;
UINT m_bulletOption;
BOOL m_switch;

```

Here, m\_enabled tracks the state of the Enable, Enable command, whereas m\_checkOption1, m\_checkOption2, and m\_checkOption3 track the commands on the Checks menu. When any of these variables are TRUE, the associated command is on. The m\_bulletOption variable holds the Bullets menu’s mode, which is the

option in the menu that's currently active. This works a little differently because, unlike the check marks in the Checks menu, only one option at a time can be active in Bullets menu. The different modes are defined in the enumeration near the top of the MAINFRM.H file, like this:

```
enum {Option1, Option2, Option3};
```

Finally, the `m_switch` variable holds the state of the Switch menu's single command, which is either the text "On" or "Off," depending on the command's state. You'll see how these variables work in conjunction with the menus and the program code later in this chapter, in the section titled "Writing Update-Command-UI Functions."

## Declaring Menu Message Handlers

As you know, MFC uses message maps to associate message response functions with messages Windows sends to the application. Some of the messages are system messages, such as `WM_PAINT`, which are sent automatically by Windows. Other messages that an application must respond to are the messages triggered when the user selects menu commands. You'll soon see how to add these messages to your message map. However, you also need to declare the message response functions with which these messages will be associated. Listing 6.6 shows how the Menu application's `CMainFrame` class declares its menu message response member functions.

### Listing 6.6 LST06\_06.CPP—Declaring Menu Message Handlers

---

```
afx_msg void OnEnableEnable();
afx_msg void OnEnableOption1();
afx_msg void OnEnableOption2();
afx_msg void OnChecksOption1();
afx_msg void OnChecksOption2();
afx_msg void OnChecksOption3();
afx_msg void OnBullets(UINT nID);
afx_msg void OnSwitchOnOff();
```

---

With the exception of the `OnBullets()` function, each of the menu message map functions are named using the same convention. The function name starts with the prefix `On` followed by the menu's title and the command's title in upper and lowercase. So, for example, the function associated with the Enable menu's Enable command is called `OnEnableEnable()`, whereas the function associated with the Checks menu's Option1 command is called `OnChecksOption1()`. The `OnBullets()` function gets its name from the Bullets menu. However, as you'll soon see, `OnBullets()` handles all of the commands on the Bullets menu, so the individual command names are not added to the message map function's name.

In most MFC applications, you'll also need to create update-command-UI functions, which are responsible for the appearance of menu commands. For example, update-command-UI functions can add check marks or bullets to

menu items. They can also enable or disable menu items. Just as with the regular menu message handlers, the update-command-UI handlers must be declared in your class. Listing 6.7 shows how they're declared in the Menu application's CMainFrame class.

### **Listing 6.7 LST06\_07.CPP—Declaring Update-Command-UI Member Functions**

---

```
afx_msg void OnUpdateEnableEnableUI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateEnableOption1UI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateEnableOption2UI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateChecksOption1UI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateChecksOption2UI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateChecksOption3UI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateBulletsUI(CCcmdUI* pCmdUI);
afx_msg void OnUpdateSwitchOnOffUI(CCcmdUI* pCmdUI);
```

---

The naming convention for update-command-UI functions is similar to that used for regular menu message handlers. The main difference is that the prefix is OnUpdate, and there's also the UI suffix. So, the name of an update-command-UI function for the Checks menu's Option2 command is OnUpdateChecksOption2UI(). All of the update-command-UI functions have a single parameter, which is a pointer to a CCmdUI object. You'll soon learn how the update-command-UI functions control the appearance of menu items.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Defining the Message Map

Now that you have your menu message map functions declared, it's time to define the message map itself. To define the various types of message map functions used with menus, you use four new message map macros:

ON\_COMMAND, ON\_COMMAND\_RANGE, ON\_UPDATE\_COMMAND\_UI, and ON\_UPDATE\_COMMAND\_UI\_RANGE. Listing 6.8 shows the message map entries for the Menu application's message handlers.

### Listing 6.8 LST06\_08.CPP—Message Handler Message Map Entries

```

ON_COMMAND( IDM_ENABLE_ENABLE, OnEnableEnable )
ON_COMMAND( IDM_ENABLE_OPTION1, OnEnableOption1 )
ON_COMMAND( IDM_ENABLE_OPTION2, OnEnableOption2 )
ON_COMMAND( IDM_CHECKS_OPTION1, OnChecksOption1 )
ON_COMMAND( IDM_CHECKS_OPTION2, OnChecksOption2 )
ON_COMMAND( IDM_CHECKS_OPTION3, OnChecksOption3 )
ON_COMMAND( IDM_SWITCH_ONOFF, OnSwitchOnOff )
    
```

Use the ON\_COMMAND macro to map a single menu command to a message response function. The macro's first argument is the message ID, and the second argument is the name of the function that will handle that message. For example, in the Menu application, the Enable, Option1 command has a command ID of IDM\_ENABLE\_OPTION1. This message ID, which is sent to the application whenever the user selects the Enable, Option1 command, is mapped to the message response function OnEnableOption1().

If you want to map a range of menu commands to a single message response function, use the ON\_COMMAND\_RANGE macro, like this:

```

ON_COMMAND_RANGE( IDM_BULLETS_OPTION1,
                  IDM_BULLETS_OPTION3, OnBullets )
    
```

The ON\_COMMAND\_RANGE macro's three arguments are the range's starting ID, the range's ending ID, and the name of the message response function. In order for this type of message mapping to work, the command IDs in the range must be consecutive. In the previous example, the Menu application's IDM\_BULLETS\_OPTION1, IDM\_BULLETS\_OPTION2, and IDM\_BULLETS\_OPTION3 command IDs (which are associated with the Option1, Option2, and Option3 commands on the Bullets menu) are all being

mapped to the `OnBullets()` message response function. When you examine `OnBullets()` later in this chapter, in the section titled “Writing Menu Message Handlers,” you’ll see how the range mapping works.

---

**Note:**

When I said that the command IDs used in the `ON_COMMAND_RANGE` macro must be consecutive, I’m referring to the values of the IDs, not the apparent order of the constants that represent the IDs. For example, if the constants `CONST1`, `CONST2`, and `CONST3` represent the values 100, 101, and 102, respectively, they are consecutive. However, if the `CONST1` equals 100, `CONST2` equals 102, and `CONST3` equals 101, they are not consecutive—even though the constant names seem to be consecutive.

---

- See “Defining a Message Map,” p. 68

Once you have your message-response functions added to your message map, it’s time to think about any update-command-UI functions you might need. If you have any menu items that are subject to enabling/disabling, check marking, or other similar changes, you’ll need to add `ON_UPDATE_COMMAND_UI` macros to your message map. In the Menu application, the update-command-UI entries look like Listing 6.9.

**Listing 6.9 LST06\_09.CPP—Defining Update-Command-UI Message Map Entries**

---

```
ON_UPDATE_COMMAND_UI( IDM_ENABLE_ENABLE ,
    OnUpdateEnableEnableUI )
ON_UPDATE_COMMAND_UI( IDM_ENABLE_OPTION1 ,
    OnUpdateEnableOption1UI )
ON_UPDATE_COMMAND_UI( IDM_ENABLE_OPTION2 ,
    OnUpdateEnableOption2UI )
ON_UPDATE_COMMAND_UI( IDM_CHECKS_OPTION1 ,
    OnUpdateChecksOption1UI )
ON_UPDATE_COMMAND_UI( IDM_CHECKS_OPTION2 ,
    OnUpdateChecksOption2UI )
ON_UPDATE_COMMAND_UI( IDM_CHECKS_OPTION3 ,
    OnUpdateChecksOption3UI )
ON_UPDATE_COMMAND_UI( IDM_SWITCH_ONOFF ,
    OnUpdateSwitchOnOffUI )
```

---

As you can see, update-command-UI message map entries are similar to the regular `ON_COMMAND` entries, requiring two arguments: the ID for which you’re defining the entry and the name of the function to which the ID should be mapped. However, although the definition of the table entries is similar, the way you write the actual response update-command-UI functions is very different, as you’ll soon see.

There is also a version of the `ON_UPDATE_COMMAND_UI` macro that can map a range of IDs to a single function. As you may have guessed, that macro is `ON_UPDATE_COMMAND_UI_RANGE`, which is used like this:

```
ON_UPDATE_COMMAND_UI_RANGE( IDM_BULLETS_OPTION1 ,  
    IDM_BULLETS_OPTION3, OnUpdateBulletsUI )
```

The macros arguments are the ID for the start of the range, the ID for the end of the range, and the name of the function to which the IDs should be mapped.

## Writing Menu Message Handlers

When it comes to responding to menu commands, the process is fairly straightforward once you've created the message map. You just write a function that performs whatever tasks the menu command is supposed to handle. For example, in the Menu application, the Enable, Enable command turns on the other commands in the Enable menu, as shown in Listing 6.10.

### Listing 6.10 LST06\_10.CPP—Responding to a Menu Command

---

```
void CMainFrame::OnEnableEnable( )  
{  
    m_enabled = !m_enabled;  
}
```

---

The OnEnable() message response function simply toggles the value of the m\_enabled flag, which determines whether the Option1 and Option2 commands in the Enable menu will be enabled or disabled. The m\_enabled flag is used in the OnUpdateEnableOption1UI(), OnUpdateEnableOption2UI(), and OnUpdateEnableEnableUI() functions to determine the visual state of the associated menu items.

Once the Enable, Option1 and Enable, Option2 commands are enabled, the user can select them, which causes MFC to call the associated message response function. In the case of Option1, that function is OnEnableOption1(), which is shown in Listing 6.11.

### Listing 6.11 LST06\_11.CPP—Responding to the Enable Menu Option1 Command

---

```
void CMainFrame::OnEnableOption1( )  
{  
    MessageBox( "Enable/Option1" );  
}
```

---



Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)

 **BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

As you can see, this function simply displays a message box, indicating to the user that the menu command was received. In a real application, you'd almost certainly want to do something a little more sophisticated. However, regardless of what the message response function does or how complex it gets, the concept is the same. The user chooses a menu command; MFC calls the associated message response function; and that function performs whatever tasks are required by the command.

What about your need to respond to a range of commands with a single message response function? You may remember that the `ON_COMMAND_RANGE` macro sets up your message map to handle this eventuality. In the Menu application, all the commands on the Bullets menu are handled by a single function, shown in Listing 6.12.

#### Listing 6.12 LST06\_12.CPP—Responding to a Range of Command IDs

```
void CMainFrame::OnBullets(UINT nID)
{
    if (nID == IDM_BULLETS_OPTION1)
        m_bulletOption = Option1;
    else if (nID == IDM_BULLETS_OPTION2)
        m_bulletOption = Option2;
    else
        m_bulletOption = Option3;
}
```

`OnBullets()` receives one parameter, `nID`, which is the ID number of the command on whose behalf the function was called. By examining `nID`, you can easily perform whatever is required of the specific command. In the `OnBullets()` function, the program sets the currently selected option by saving it in `m_bulletOption`. The `m_bulletOption` variable not only informs the program of which option is currently active, but also, as you'll soon see, enables the appropriate update-command-UI function to properly update the visual state of the Bullets menu.

## Writing Update-Command-UI Functions

The last topic you need to cover in this chapter is the update-command-UI functions, which control how your menu commands look to the user when a pop-up menu is displayed. As with the more conventional message response functions, how you write your update-command-UI functions depends upon each menu's purpose. In the case of the Menu application's Checks menu, the function only

needs to check or uncheck the menu command, depending upon the value of the flag associated with the command. For example, look at the `OnUpdateChecksOption1UI()` function shown in Listing 6.13.

---

**Listing 6.13 LST06\_13.CPP—Updating the Appearance of Menu Commands**

---

```
void CMainFrame::OnUpdateChecksOption1UI(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_checkOption1);
}
```

---

Here you can see that the function receives a single parameter, which is a pointer to a `CCmdUI` object. You use the `CCmdUI` object's member functions to manipulate the menu command. These member functions are `Enable()` (which enables or disables a menu item), `SetCheck()` (which adds or removes a check mark), `SetRadio()` (which adds or removes bullets), and `SetText()` (which changes a menu command's caption). All of these functions, except `SetText()`, require a single parameter. A non-zero value turns the attribute on, and a 0 turns the attribute off. For example, calling `SetCheck(TRUE)` turns a check mark on, whereas calling `SetCheck(FALSE)` turns it off. The argument for the `SetText()` function is the new text string for the menu item.

In Listing 6.13, the `m_checkOption1` flag is either `TRUE` or `FALSE`, so it can be used to directly control whether the menu command displays a check mark. If you recall, the value of `m_checkOption1` is toggled whenever the user selects the Checks, Option1 command. The entire process goes something like this:

1. The user clicks the Checks menu caption in order to open the Checks pop-up menu.
2. Before displaying the menu commands in the Checks menu, MFC calls the `OnUpdateChecksOption1UI()`, `OnUpdateChecksOption2UI()`, and `OnUpdateChecksOption3UI()` functions. Because the `m_checksOption1`, `m_checksOption2`, and `m_checksOption3` flags start off initialized to `FALSE`, the calls to `SetCheck()` in the update-command-UI functions add no check marks to the menu items.
3. The Checks pop-up menu appears, and the user clicks Option1.
4. MFC calls the `OnChecksOption1()` message response function, which toggles the `m_checksOption1` flag from `FALSE` to `TRUE`.
5. The Checks menu closes.
6. The user clicks the Checks menu again.
7. Before displaying the Checks menu's commands, MFC again calls each of the update-command-UI functions associated with the commands. This time, the call to `OnUpdateChecksOption1UI()` places a check mark next to the Option1 command because the `m_checksOption1` flag, which is used as the argument for the `SetCheck()` function call, is now `TRUE`.

A similar process to the one just outlined occurs every time the user opens a menu, with the update-command-UI functions updating each menu command's appearance before they are displayed.

Of course, as you may be thinking, the Checks menu commands might be better

handled by a single update-command-UI function. And, you're right. The second, and more elegant, solution is demonstrated in the Menu application by the OnUpdateBulletsUI() function, which handles the range of message IDs from IDM\_BULLETS\_OPTION1 through IDM\_BULLETS\_OPTION3, as shown in Listing 6.14.

#### **Listing 6.14 LST06\_14.CPP—Updating a Range of Menu Commands**

---

```
void CMainFrame::OnUpdateBulletsUI(CCmdUI* pCmdUI)
{
    if (pCmdUI->m_nID == IDM_BULLETS_OPTION1)
        pCmdUI->SetRadio(m_bulletOption == Option1);
    else if (pCmdUI->m_nID == IDM_BULLETS_OPTION2)
        pCmdUI->SetRadio(m_bulletOption == Option2);
    else
        pCmdUI->SetRadio(m_bulletOption == Option3);
}
```

---

The function in Listing 6.14 gets called three times when the user opens the Bullets menu: once for each command in the menu. The CCmdUI object passed as the function's single parameter contains a data member, m\_nID, that holds the ID of the message for which the function is currently being called. By using this ID—as well as the value of m\_bulletOption, which holds the currently selected option—the function can easily add or remove the bullets from the menu commands.

MFC gives you all the tools you need to create professional-looking menus for your applications. By taking advantage of message maps, you can associate a specific function with a menu message as well as keep your menu commands visually updated. Menus are an important element of most Windows applications—an element that takes some work to implement properly. However, menu handling in an MFC program is much simpler than writing your menu-handling code from scratch.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Chapter 7

# Programming Dialog Boxes

- How to create a dialog box resource
- How to display a dialog box
- How to write a custom dialog box class
- How to extract information from a dialog box
- How to validate the contents of a dialog box's controls

If it weren't for communication, we would all be living in a lonely world indeed. With no way to transfer our thoughts and ideas to other people, our minds would be sealed traps, forever locking up all the knowledge we managed to acquire over our lifetimes. Luckily, we have, over the years, managed to solve many of our communications problems. But these solutions were a long time coming. Way, way back, early humans communicated with body gestures. Then language came along, enabling the transference of more sophisticated ideas. When machines were invented, so were ways to communicate with them, including such devices as patch bays and keyboards.

What has this to do with MFC? When Windows and other graphically oriented user interfaces came along, programmers needed to find a new way to retrieve information from the computer user—a communication problem for the modern age. In the days of DOS, the program could simply print a prompt on-screen and direct the user to enter whatever value the program needed. With Windows, however, getting data from the user is not so simple. Because a window's client area should be reserved for other purposes, a well designed Windows application gets most user input through dialog boxes.

MFC includes several types of dialog boxes. In this chapter, you learn to use MFC to display and handle your own custom dialog boxes, which you design with Developer Studio's dialog box editor.

## Understanding Dialog Boxes

As mentioned in the introduction to this chapter, dialog boxes are one way a Windows application can get information from the user. Chances are that your Windows application will have several dialog boxes, each designed to retrieve a different type of information from your user. However, before you can add these dialog boxes to your program, you must create them. To make this job

simpler, Developer Studio includes an excellent resource editor. In Chapter 6, “Using Menus,” you used the resource editor to create menus; you can use this handy visual tool to create all types of resources, including bitmaps, icons, string tables, and, of course, dialog boxes.

- **See “Creating a Menu Resource,” p. 94**

Because dialog boxes are used so extensively in Windows applications, MFC provides several classes that you can use to make dialog box manipulation easier and more convenient. Although you can use MFC’s `CDialog` class directly, you’ll most often derive your own dialog box class from `CDialog` in order to have more control over the way that your dialog box operates. However, MFC also provides classes for more specific types of dialog boxes, including `CColorDialog`, `CFontDialog`, and `CFileDialog`.

The minimum steps for adding a dialog box to your MFC application are as follows:

1. Create your dialog box resource using the Visual C++ dialog box editor. This process defines the appearance of the dialog box, including the types of controls it will contain.
2. Create an instance of `CDialog`, passing to the constructor your dialog box’s resource ID and a pointer to the parent window.
3. Call the dialog box object’s `DoModal()` member function to display the dialog box.

Although the preceding steps are all you need to display a simple dialog box, you’ll usually need much more control over your dialog box than these steps provide. For example, this method of displaying a dialog box enables no way to retrieve data from the dialog box, which means the method is really only useful for dialog boxes that display information to the user—sort of like a glorified message box. To create a dialog box that you can control, perform the following steps:

1. Create your dialog box resource using the Visual C++ dialog box editor. This process defines the appearance of the dialog box as well as the controls that appear in the dialog box and the types of data those controls will return to your program.
2. Write a dialog box class derived from `CDialog` for your dialog box, including member variables for storing the dialog box’s data. Initialize the member variables in the class’s constructor.
3. Overload the `DoDataExchange()` function in your dialog box class. In the function, call the appropriate DDX and DDV functions to perform data transfer and validation.
4. In the window class that’ll display the dialog box, create member variables for the dialog box controls whose data you want to store.
5. Create an instance of your dialog box class.
6. Call the dialog box object’s `DoModal()` member function to display the dialog box.
7. When `DoModal()` returns, copy the dialog box data that you need to store from the dialog box object’s member variables to the window class’s matching variables.

In the rest of this chapter, you'll see how to perform all of the steps listed previously, including how to write your dialog box class and how to provide this class with automatic data transfer and validation. In the next section, you learn to use Developer Studio's dialog box editor.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Creating a Dialog Box Resource

As you now know, the first step toward adding a dialog box to your MFC application is creating the dialog box resource, which acts as a sort of template for Windows. When Windows sees the dialog box resource in your program, it uses the commands in the resource to construct the dialog box for you. To learn how to create a dialog box resource, just perform the following steps:

1. Choose the Insert, Resource from the Developer Studio's menu bar. The InsertResource dialog box appears, as shown in Figure 7.1.



**FIG. 7.1** The Insert Resource dialog box enables you to select the type of resource you want to add to your project.

2. Double-click Dialog in the Resource Type box. The dialog box editor appears in one of the Developer Studio's panes (see Figure 7.2).



**FIG. 7.2** Use the dialog box editor to create your dialog box template by placing and editing the controls provided in the editor's toolbox.

3. Select controls on the toolbox and position them on the window of your dialog box, as shown in Figure 7.3.



**FIG. 7.3** The dialog box editor's toolbox provides access to the controls you can place on your dialog box.

4. Double-click the controls you want to edit. The control's property sheet appears, in which you can supply your own ID, caption, and other control information (see Figure 7.4).



**FIG. 7.4** Each control has its own property sheet in which you can enter the information needed to customize the control for use in your dialog box.

5. Select the dialog box, and then press Enter. The dialog box's property sheet, which enables you to customize the dialog box attributes, appears (see Figure 7.5).





**FIG. 7.5** Just like controls, a dialog box has a property sheet that you can customize.

## Defining Dialog Box and Control IDs

You might remember that, when you created your menu resource in Chapter 6, you were able to choose menu IDs from predefined system IDs or create your own custom IDs. Because dialog boxes are often unique to an application (with the exception of the common dialog boxes), you will almost always create your own IDs for both the dialog box and the controls it contains. You can, if you like, accept the default IDs that the dialog box editor creates for you. However, these IDs are generic (i.e. `IDD_DIALOG1`, `IDC_EDIT1`, `IDC_RADIO1`, and so on), and so you'll probably want to change them to something more specific.

In any case, as you can tell from the default IDs, a dialog box ID usually begins with the prefix `IDD_` and control IDs usually begin with the prefix `IDC_`. You can, of course, use your own prefixes if you like, although sticking with conventions often makes your programs easier to read.

## Looking at the Resource Script of a Dialog Box

As you learned in Chapter 6, "Using Menus," after creating a resource, Developer Studio creates or modifies at least two files that you need to add to your application. If you remember, the first file is called `RESOURCE.H` and contains the resource IDs that you've defined. You must include `RESOURCE.H` in any file that refers to these IDs.

- See "Dealing with Resource Files," p. 97

The second file Developer Studio creates or modifies has the `.RC` extension and is the resource script that defines all your application's resources. In Chapter 6, you learned that the resource script is like a source code file written in a language that the resource compiler understands. The resource compiler takes the `.RC` file and compiles it into a `.RES` file, which is the binary representation of your application's resources. Listing 7.1 shows part of a resource script that defines a dialog box.

### Listing 7.1 LST07\_01.RC—The Resource Script that Defines a Dialog Box

```
IDD_DIALOG1 DIALOG DISCARDABLE 0, 0, 203, 151
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON    "OK", IDOK, 7, 130, 91, 14
    PUSHBUTTON       "Cancel", IDCANCEL, 105, 130, 91, 14
    LTEXT            "Name: ", IDC_STATIC, 14, 14, 23, 10
    EDITTEXT         IDC_NAME, 39, 13, 143, 12, ES_AUTOHSCROLL
    COMBOBOX         IDC_DEPARTMENT, 13, 74, 80, 72, CBS_DROPDOWN | CBS_SORT |
        WS_VSCROLL | WS_TABSTOP
    LTEXT            "Department: ", IDC_STATIC, 14, 62, 41, 10
    GROUPBOX         "Availability", IDC_STATIC, 99, 33, 85, 87
    CONTROL          "Monday", IDC_MONDAY, "Button", BS_AUTOCHECKBOX |
        WS_TABSTOP, 119, 47, 60, 10
    CONTROL          "Tuesday", IDC_TUESDAY, "Button", BS_AUTOCHECKBOX |
        WS_TABSTOP, 119, 60, 63, 10
    CONTROL          "Wednesday", IDC_WEDNESDAY, "Button", BS_AUTOCHECKBOX |
        WS_TABSTOP, 119, 73, 56, 11
    CONTROL          "Thursday", IDC_THURSDAY, "Button", BS_AUTOCHECKBOX |
        WS_TABSTOP, 119, 87, 49, 10
```

```
CONTROL          "Friday", IDC_FRIDAY, "Button", BS_AUTOCHECKBOX |
                  WS_TABSTOP, 119, 100, 38, 9
LTEXT             "Age: ", IDC_STATIC, 15, 41, 17, 9
EDITTEXT          IDC_AGE, 32, 38, 32, 15, ES_AUTOHSCROLL
END
```

---

If you examine Listing 7.1, even though you might not yet understand the script language, you can easily see hierarchical organization of the dialog box and its controls. Also, you can see that each control's definition includes a number of values including its caption, ID, style, position, and size. When you use Developer Studio's resource editors to create your resources, you don't usually need to modify the resultant resource script directly. However, there might be times when such direct editing will be convenient, so it couldn't hurt to learn more about how resource scripts work. Consult your favorite Windows programming manual for more information on resource scripts.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full
 

- Advanced
- Search
- Search Tips

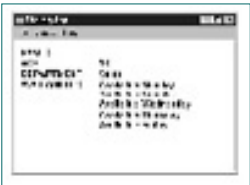
[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Introducing the Dialog Application



This chapter’s sample program, called Dialog, demonstrates two ways to create and display dialog boxes in your Windows applications. You can find the program, along with all its source code, in the CHAP07\DIALOG folder of this book’s CD-ROM (or on your hard drive if you installed the contents of the CD-ROM). To run the program, double-click the DIALOG.EXE file. When you do, you see the window shown in Figure 7.6.



**FIG. 7.6** The Dialog application’s main window displays the data collected from one of its dialog boxes.

The data displayed in the window are the default values for information that can be collected by one of the program’s dialog boxes. You’ll take a look at that dialog box in a second, but first select the Help, About command to display the application’s About dialog box. As you can see in Figure 7.7, the About dialog box only displays information to the user and does not enable the user to enter data. For this reason, the About dialog box can be created and displayed very easily in an MFC program, as you’ll soon see.



**FIG. 7.7** The About dialog box doesn’t interact with the user (except through the OK button), so it’s easy to create and display in the program.

To display the application’s second—and much more complex—dialog box, select the Dialog, Test command. When you do, a dialog box containing a number of controls appears. You can use these controls to enter information about an imaginary employee (see Figure 7.8). When you finish entering information, close the dialog box by clicking OK. The Dialog application’s main window then displays the new information you entered into the dialog box’s controls (see Figure 7.9).



**FIG. 7.8** The Dialog, Test command reveals a dialog box containing several types of controls.



**FIG. 7.9** After transferring data from the dialog box, you can display the user's choices in the window.

This second dialog box is also capable of validating its data. For example, the Name text box will not let you enter more than 30 characters. In addition, the Age text box will reject any value that doesn't fall between 16 and 100. If you enter an invalid value into the Age text box, you will see the warning shown in Figure 7.10, and you will not be allowed to exit the dialog box with the OK button until the value is corrected. Notice also that, if you enter new data into the dialog box and then select the Cancel button, your changes are thrown away and do not appear in the application's window.



**FIG. 7.10** Dialog boxes in MFC programs can validate their own data.

## Exploring the Dialog Application

Now that you've had a chance to use the Dialog application, it's time to see how it works. Listing 7.2 through Listing 7.5 are the most pertinent source code files for the Dialog application. Listing 7.2 and Listing 7.3 constitute the application's CMainFrame class, which, of course, represents the application's main window. Listing 7.4 and Listing 7.5 are the source code files for the CDlg1 class, which represents the dialog box that appears when you select the application's Dialog, Test command. Due to its simplicity, the program doesn't need a special class for the About dialog box. Also, because there are few changes to the application class as compared with other programs in this book, the DIALOG.H and DIALOG.CPP files, which are the source code for the CDialogApp class, are not shown here.

### Listing 7.2 MAINFRM.H—The Header File of the Main Window Class

---

```

////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//             represents the application's main window.
////////////////////////////////////

```

```

class CMainFrame : public CFrameWnd
{
// Protected data members.
protected:
    CString m_name;
    UINT m_age;
    CString m_department;
    BOOL m_monday;
    BOOL m_tuesday;
    BOOL m_wednesday;
    BOOL m_thursday;
    BOOL m_friday;

// Constructor and destructor.
public:
    CMainFrame();
    ~CMainFrame();

// Overrides.
protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Message map functions.
protected:
    // System message handlers.
    afx_msg void OnPaint();

    // Menu command message handlers.
    afx_msg void OnDialogTest();
    afx_msg void OnHelpAbout();

    // Update command UI handlers.
    // None.

// Protected member functions.
protected:

    DECLARE_MESSAGE_MAP()
};

```

---

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Listing 7.5 DLG1.CPP—The Implementation File of the CDlg1 Class

```

////////////////////////////////////
// DLG1.CPP: Implementation file for the CDLG1 class.
////////////////////////////////////

#include <afxwin.h>
#include "dialog.h"
#include "resource.h"
#include "dlg1.h"

////////////////////////////////////
// CONSTRUCTOR
////////////////////////////////////
CDlg1::CDlg1(CWnd* pParent) : CDialog(IDD_TESTDIALOG, pParent)
{
    // Initialize data transfer variables.
    m_department = "Sales";
    m_monday = TRUE;
    m_tuesday = TRUE;
    m_wednesday = TRUE;
    m_thursday = TRUE;
    m_friday = TRUE;
    m_name = "";
    m_age = 16;
}

////////////////////////////////////
// Overrides.
////////////////////////////////////
void CDlg1::DoDataExchange(CDataExchange* pDX)
{
    // Call the base class's version.
    CDialog::DoDataExchange(pDX);

    // Associate the data transfer variables with
    // the ID's of the controls.
    DDX_Text(pDX, IDC_NAME, m_name);
    DDV_MaxChars(pDX, m_name, 30);
    DDX_Text(pDX, IDC_AGE, m_age);
    DDV_MinMaxUInt(pDX, m_age, 16, 100);
}

```

```
DDX_CBString(pDX, IDC_DEPARTMENT, m_department);
DDX_Check(pDX, IDC_MONDAY, m_monday);
DDX_Check(pDX, IDC_TUESDAY, m_tuesday);
DDX_Check(pDX, IDC_WEDNESDAY, m_wednesday);
DDX_Check(pDX, IDC_THURSDAY, m_thursday);
DDX_Check(pDX, IDC_FRIDAY, m_friday);
}
```

---

## Displaying a Simple Dialog Box

Before you get too deep into learning how to create sophisticated dialog boxes, it might be nice to see how easy it can be to display simple dialog boxes that require no complex interaction with the user. A good example is the Dialog application's About dialog box, which does nothing more than display program information to the user. The hardest part of getting the About dialog box up on the screen is creating the dialog box template with the resource editor—and that task is just a matter of positioning a few static-text controls. If you look at the `OnHelpAbout()` function of the `CMainFrame` class, you see that displaying the dialog box requires only two lines of code, as follows:

```
CDialog dlg(IDD_ABOUTDIALOG, this);
dlg.DoModal();
```

The first line creates a `CDialog` object that's associated with the About dialog box template that was defined in the resource editor. The constructor's two arguments are the dialog box's resource ID and a pointer to the dialog box's parent window, which is the `CMainFrame` window. After the dialog box object has been constructed and associated with the template, a call to the dialog box object's `DoModal()` member function displays the dialog box on the screen.

The `DoModal()` function handles all the user's interactions with the dialog box, which, in this case, amounts to little more than waiting for the user to click the OK button. When the user exits the dialog box, the `DoModal()` function returns, and your program can continue. Because the dialog box object is created locally on the stack, it is automatically deleted when it goes out of scope, which is when the `OnHelpAbout()` function exits. You could also create the dialog box dynamically on the heap, by using the `new` operator. In that case, you'd have to delete the object yourself. The following lines illustrate this technique:

```
CDialog* dlg = new CDialog(IDD_ABOUTDIALOG, this);
dlg->DoModal();
delete dlg;
```

---

### Caution

When creating your dialog box's template with the resource editor, make sure that you turn on the dialog box's `Visible` attribute, which is found in the Dialog's property sheet on the `More Styles` page. If you fail to set this style attribute, your dialog box will not appear on the screen properly, making the application seem to lock up.

---

## Writing a Dialog Box Class

Displaying a simple About dialog box is all well and good. Unfortunately though, most



dialog boxes are much more complex, requiring that your application be able to transfer information back and forth between the program and the dialog box. Often, you'll want to initialize the dialog box's controls with default values before displaying the dialog box. After the user enters information into the dialog box, you then need to extract that information so that it can be used in your program. Accomplishing this means not only creating a dialog box template with the resource editor, but also associating that template with your own custom dialog box class derived from MFC's CDialog.

Figure 7.11 shows the Test Dialog from the Dialog application under construction in Developer Studio's dialog box editor. As you can see, the dialog box has ten controls that the user can manipulate. These controls are the Name text box, the Age text box, the Department combo box, the five check boxes for the days of the week, and the OK and Cancel buttons.



**FIG. 7.11** As with most dialog boxes, the Dialog application's dialog box was created using Developer Studio's resource editors.

You don't have to worry about the OK and Cancel buttons because MFC handles them for you. When you write your dialog box's class, however, you do need to provide data members for storing the information the user enters into the other controls. The CDialog1 class, which is derived from CDialog, provides the needed data members, as shown in Listing 7.6.

---

**Listing 7.6 LST07\_06.CPP—Declaring Data Members for the Dialog Box Controls**

---

```
public:
    CString m_department;
    BOOL m_monday;
    BOOL m_tuesday;
    BOOL m_wednesday;
    BOOL m_thursday;
    BOOL m_friday;
    CString m_name;
    UINT m_age;
```

---

Notice that these variables are declared as public data members of the class. This is important because if they were declared as private or protected, your program would not be able to access them outside of the class. (Yes, I know, this is another case where MFC breaks the rules of strict object-oriented design, which dictates that a class's data member should never be accessed from outside of the class.)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 7.1 DDX Functions**

Function	Description
DDX_CBIndex()	Links a combo box's index to an integer variable
DDX_CBString()	Links a combo box's string to a string variable
DDX_CBStringExact()	Links a combo box's selected string to a string variable
DDX_Check()	Links a check box with an integer variable
DDX_LBIndex()	Links a list box's index with an integer variable
DDX_LBString()	Links a list box's string to a string variable
DDX_LBStringExact()	Links a list box's selected string to a string variable
DDX_Radio()	Links a radio button with an integer variable
DDX_Scroll()	Links a scroll bar to an integer variable
DDX_Text()	Links a text box to a string variable

MFC's DDV functions perform data validation for controls. For example, in Listing 7.9, the call to `DDV_MaxChars()` tells MFC that a valid entry in the edit control linked to `m_name` should be no longer than thirty characters. After the text box control reaches 30 characters, it will accept no more input from the user. On the other hand, the call to `DDV_MinMaxInt()` tells MFC that the `m_age` variable, which is associated with the edit control whose ID is `IDC_AGE`, should not accept values outside the range of 16 to 100. If the user enters an invalid number in the `IDC_AGE` edit box, MFC displays a message, warning the user of his mistake. The user cannot exit via the OK button of the dialog box until the value is corrected. Table 7.2 shows the DDV functions you can call. For more information on their parameters, consult your Visual C++ online documentation.

**Table 7.2 DDV Functions**

Function	Description
DDV_MaxChars()	Limits the length of a string
DDV_MinMaxByte()	Limits a byte value to a specific range
DDV_MinMaxDouble()	Limits a double value to a specific range
DDV_MinMaxDWord()	Limits a DWORD value to a specific range
DDV_MinMaxFloat()	Limits a floating-point value to a specific range
DDV_MinMaxInt()	Limits an integer value to a specific range
DDV_MinMaxLong()	Limits a long integer value to a specific range
DDV_MinMaxUInt()	Limits an unsigned integer value to a specific range

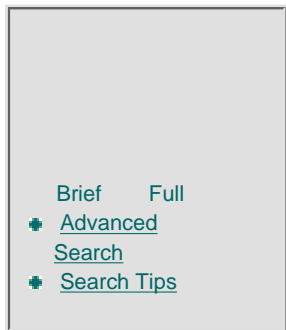
**Note**

It's important that you call DDV functions immediately after the DDX function that sets up the data exchange for a control. When you do this, MFC can set the input focus to the control that contains invalid data—not only showing the user exactly where the problem is, but also enabling him to enter a new value

## Using the Dialog Box Class

Now that you have your dialog box class written, you can create objects of that class within your program and display the associated dialog box element. The first step in using your new class is to include the header file of the dialog box class in any class that'll access the class. Failure to do this will cause the compiler to complain that it doesn't recognize the dialog box class. Also, when you develop the window class that'll display the dialog box, you'll usually want to create data members that mirror the data members of the dialog box class. This gives you a place to store information that you transfer from the dialog box. The CMainFrame class declares this set of member variables, as shown in Listing 7.10.

[Previous](#) [Table of Contents](#) [Next](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

What are DDX and DDV functions? An excellent question! MFC's DDX functions set up the link between a data member and a control. For example, in Listing 7.9, the call to DDX\_Text() links the edit control whose ID is IDC\_NAME to the data member m\_name. This tells MFC to copy the contents of m\_name to the control when the dialog box is displayed and to copy the data in the control back to m\_name when the dialog box is dismissed. Notice that a DDX function's first argument is the CDataExchange pointer passed into the DoDataExchange() function. There is a DDX function for each type of control you might want to include in a data transfer. These functions are listed in Table 7.1. To learn about each function's parameters, look them up in your Visual C++ online documentation.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 8

# Programming Controls

- About the different types of Windows controls
- How to associate an MFC class with a control
- How to call control class member functions
- How to write a custom data transfer and validation system

In a Windows application, dialog boxes are the most common object used to get information from the user to the program. Of course, dialog boxes aren't much good without the many controls that can be placed in them. Edit controls, list boxes, combo boxes, radio buttons, check boxes, and other types of controls all work together to provide the application's user with convenient ways to enter data into a program. Without these controls, a dialog box is about as useful as a telephone without number buttons.

In this chapter, you learn to program window controls in dialog boxes. You'll also learn new ways to extract information from a window's controls.

## Understanding Controls

Until now, the programs in this book have featured dialog boxes containing basic controls such as edit controls and buttons. Edit controls and buttons are probably the most important types of controls at your disposal. However, often you can give your program's user easier methods of selecting the data that must be entered into the program.

There are several types of controls you can place in a dialog box, including check boxes, radio buttons, list boxes, combo boxes, and scroll bars. You should already be familiar with how these controls work, both from a user's and a programmer's point of view. If you've never programmed with Microsoft Foundation Classes, you may not be familiar with the classes with which MFC encapsulates each of the window controls. These classes include CButton, CEdit, CStatic, CListBox, and CComboBox.

Although MFC supplies classes that encapsulate the many window controls, Windows provides most of the services needed to handle these controls. A description of each window control follows:

- **Static text** Static text is a string of characters usually used to label other controls in a dialog box or window. Although it is considered to be

a window control, static text cannot be manipulated by the user. You use the CStatic class to create and manipulate static text.

- **Edit box** An edit control accepts text input from the user. The user can edit the text in various ways before completing the input. You use the CEdit class to create and manipulate an edit box.
- **Pushbutton** A pushbutton (also simply called a “button”) is a graphical object that triggers a command when the user clicks it. When clicked, a button’s graphical image is usually animated to appear as if the button is pressed and released. You use the CButton class to create and manipulate pushbuttons.
- **Check box** A check box is a special type of button that toggles a check mark when clicked. Check boxes usually represent program options that the user can select. You use the CButton class to create and manipulate check boxes.
- **Radio button** Radio buttons are similar to check boxes, except only one radio button in a group can be selected at a given time. Radio buttons usually represent program options that are mutually exclusive. You use the CButton class to create and manipulate radio buttons.
- **Group box** Often, check boxes and radio buttons are placed into group boxes, which organize the buttons into logical groups. The user cannot interact with group boxes. You use the CStatic class to create and manipulate group boxes.
- **List box** A list box is a rectangle containing a set of selections. These selections are usually text items but can also be bitmaps or other objects. Depending on the list box’s style flags, the user may select one or several objects in the list box. You use the CListBox class to create and manipulate a list box control.
- **Combo box** A combo box is similar to a list box, except it also includes an edit control in which the user can type a selection. You use the CComboBox class to create and manipulate combo boxes.
- **Scroll bar** A scroll bar is a graphical object containing a track that encloses a sliding box called the scroll box. By positioning the scroll box, the user can select a value from a given range. In addition to the scroll box, a scroll bar contains arrow boxes that, when clicked, move the scroll box a unit in the direction of the arrow. Although scroll bars are rarely used in dialog boxes, they can be created and manipulated by the CScrollBar class.

As mentioned previously, you can add any of these controls to a dialog box simply by using Developer Studio’s dialog box editor. Figure 8.1 shows the dialog box under construction in this chapter’s first program, Control1.



**FIG. 8.1** The dialog box under construction in Developer Studio’s resource editor contains several standard Windows controls.

## Introducing the Control1 Application

This chapter's sample program is called Control1. You can find the source code and executable file for this program in the CHAP08\CONTROL1 folder on this book's CD-ROM. When you run Control1, you see the window shown in Figure 8.2. This window shows the data that the user has currently selected from the application's dialog box. At the start of the program, this data is set to default values. The EDIT CONTROL field of the display shows the current string copied from the dialog box's edit control. The RADIO BUTTON field shows which radio button in the dialog box is selected. The CHECK BOX fields show the status of each of the three check boxes. The LIST BOX field shows the string currently selected in the list box, and the COMBO BOX field shows the currently selected string in the combo box.



**FIG. 8.2** The Control application's dialog box shows the controls' settings.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

To find out where all these controls are, select the Dialog, Test command to bring up the Control Dialog dialog box shown in Figure 8.3. This dialog box contains the controls whose data is shown in the main window. You can change the controls' settings any way you want. When you exit the Control Dialog dialog box via the OK button, the main window displays the controls' new settings. Exiting the dialog box by clicking the Cancel button has no effect on the main window's display, because any changes made to the data in the dialog box are ignored.



**FIG. 8.3** The Control1 application's dialog box enables you to manipulate several standard Windows controls.

## Exploring the Control1 Application

As you experimented with the Control1 application, it may have seemed to you that the program doesn't do a heck of a lot more than the dialog box sample you created in chapter 7, "Programming Dialog Boxes." The truth is, it's not so much what the Control1 application does, but rather how it does it. Specifically, the controls in the application's dialog box are associated with MFC control classes so that the program can directly manipulate the controls. In this section, you'll see how this bit of MFC trickery is accomplished.

Listings 8.1 through 8.4 are the most pertinent source code files for the Control1 application. Listings 8.1 and 8.2 comprise the application's CMainFrame class, which represents the application's main window.

### Listing 8.1 MAINFRM.H—The Header File for the Main Window Class

```

////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//             represents the application's main window.
////////////////////////////////////

class CMainFrame : public CFrameWnd
{
// Protected data members.
protected:
    CString m_edit1;
    int m_radiol;
    int m_radio2;
    int m_radio3;
    CString m_combol;

```



```

        int m_check1;
        int m_check2;
        int m_check3;
        CString m_list1;

// Constructor and destructor.
public:
    CMainFrame();
    ~CMainFrame();

// Overrides.
protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Message map functions.
protected:
    // System message handlers.
    afx_msg void OnPaint();

    // Menu command message handlers.
    afx_msg void OnDialogTest();

    // Update command UI handlers.
    // None.

// Protected member functions.
protected:

    DECLARE_MESSAGE_MAP()
};

```

---

### **Listing 8.2 MAINFRM.CPP—The Implementation File for the Main Window Class**

---

```

////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//               class, which represents the application's
//               main window.
////////////////////////////////////

#include <afxwin.h>
#include "mainfrm.h"
#include "resource.h"
#include "cntldlg.h"

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    // Message map entries for system messages.
    ON_WM_PAINT()

    // Message map entries for menu commands.
    ON_COMMAND(IDM_DIALOG_TEST, OnDialogTest)

    // Message map entries for update command UI handlers.

```

```

        // None.
END_MESSAGE_MAP()

////////////////////////////////////
// CMainFrame: Construction and destruction.
////////////////////////////////////
CMainFrame::CMainFrame()
{
    // Create the main frame window.
    Create(NULL, "Control App", WS_OVERLAPPEDWINDOW, rectDefault,
        NULL, MAKEINTRESOURCE(IDR_MENU1));

    // Initialize the class's data members.
    m_edit1 = "Default";
    m_radiol = 1;
    m_radio2 = 0;
    m_radio3 = 0;
    m_combol = "ComboString1";
    m_check1 = 1;
    m_check2 = 0;
    m_check3 = 0;
    m_list1 = "ListString1";
}

CMainFrame::~CMainFrame()
{
}

////////////////////////////////////
// Overrides.
////////////////////////////////////
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Set the size of the main window.
    cs.cx = 330;
    cs.cy = 200;

    // Call the base class's version.
    BOOL returnCode = CFrameWnd::PreCreateWindow(cs);

    return returnCode;
}

////////////////////////////////////
// Message map functions.
////////////////////////////////////
void CMainFrame::OnPaint()
{
    // Create a device context.
    CPaintDC paintDC(this);

    // Have Windows fill in a TEXTMETRIC structure.
    TEXTMETRIC textMetric;

```

```

paintDC.GetTextMetrics(&textMetric);

// Initialize the vertical position of a text line.
UINT position = 10;

// Display the edit control's contents.
paintDC.TextOut(10, position, "EDIT CONTROL:");
paintDC.TextOut(150, position, m_edit1);

// Display the state of the radio buttons.
position += textMetric.tmHeight;
paintDC.TextOut(10, position, "RADIO BUTTON:");
if (m_radiol)
    paintDC.TextOut(150, position, "Button #1 chosen");
else if (m_radio2)
    paintDC.TextOut(150, position, "Button #2 chosen");
else
    paintDC.TextOut(150, position, "Button #3 chosen");

// Display the selected combo box string.
position += textMetric.tmHeight;
paintDC.TextOut(10, position, "COMBO BOX:");
paintDC.TextOut(150, position, m_combol);

// Display the status of the check boxes.
position += textMetric.tmHeight;
paintDC.TextOut(10, position, "CHECK BOXES:");

if (m_check1)
    paintDC.TextOut(150, position, "Check 1 selected");
else
    paintDC.TextOut(150, position, "Check 1 not selected");
position += textMetric.tmHeight;

if (m_check2)
    paintDC.TextOut(150, position, "Check 2 selected");
else
    paintDC.TextOut(150, position, "Check 2 not selected");

position += textMetric.tmHeight;
if (m_check3)
    paintDC.TextOut(150, position, "Check 3 selected");
else
    paintDC.TextOut(150, position, "Check 3 not selected");

// Display the selected list box string.
position += textMetric.tmHeight;
paintDC.TextOut(10, position, "LIST BOX:");
paintDC.TextOut(150, position, m_list1);
}

void CMainFrame::OnDialogTest()
{

```

```
// Create and display the dialog box.
CCntldlg dialog(this);
int result = dialog.DoModal();

if (result == IDOK)
{
    // Save the contents of the dialog box.
    m_edit1 = dialog.m_edit1;
    m_radio1 = dialog.m_radio1;
    m_radio2 = dialog.m_radio2;
    m_radio3 = dialog.m_radio3;
    m_combol = dialog.m_combol;
    m_check1 = dialog.m_check1;
    m_check2 = dialog.m_check2;
    m_check3 = dialog.m_check3;
    m_list1 = dialog.m_list1;

    // Force the window to repaint.
    Invalidate();
}
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Listings 8.3 and 8.4 are the source code files for the CCntlDlg class, which represents the dialog that appears when you select the application’s Dialog, Test command. Because there are few changes to the application class as compared with other programs in this book, the CONTROL1.H and CONTROL1.CPP files, which are the source code for the CControl1App class, are not shown here.

**Listing 8.3 CNTLDLG.H—The Header File for the Dialog Box Class**

```

////////////////////////////////////
// CNTLDLG.H: Header file for the CCntlDlg class.
////////////////////////////////////

class CMainFrame;

class CCntlDlg : public CDialog
{
// Class data members.
protected:
    CString m_edit1;
    int m_radiol;
    int m_radio2;
    int m_radio3;
    int m_check1;
    int m_check2;
    int m_check3;
    CString m_list1;
    CString m_combol;

// Constructor.
public:
    CCntlDlg(CWnd* pParent);

// Overrides.
protected:
    virtual BOOL OnInitDialog();
    virtual void OnOK();

    friend CMainFrame;
};
  
```

## Listing 8.4 CNTLDLG.CPP—The Implementation File for the Dialog Box Class

---

```
////////////////////////////////////
// CNTLDLG.CPP: Implementation file for the CCtrlDlg class.
////////////////////////////////////

#include <afxwin.h>
#include "resource.h"
#include "cntldlg.h"

////////////////////////////////////
// CONSTRUCTOR
////////////////////////////////////
CControlDlg::CControlDlg(CWnd* pParent) :
    CDialog(IDD_CONTROLDIALOG, pParent)
{
}

////////////////////////////////////
// Overrides.
////////////////////////////////////
BOOL CControlDlg::OnInitDialog()
{
    // Make sure the base class gets initialized properly.
    CDialog::OnInitDialog();

    // Set the text in the edit box.
    CEdit* pEdit1 = (CEdit*)GetDlgItem(IDC_EDIT1);
    pEdit1->SetWindowText("Default");

    // Select the first radio button.
    CButton* pRadio1 = (CButton*)GetDlgItem(IDC_RADIO1);
    pRadio1->SetCheck(TRUE);

    // Select the first check box.
    CButton* pCheck1 = (CButton*)GetDlgItem(IDC_CHECK1);
    pCheck1->SetCheck(TRUE);

    // Add strings to the list box and select the
    // first string in the list.
    CListBox* pList1 = (CListBox*)GetDlgItem(IDC_LIST1);
    pList1->AddString("ListString1");
    pList1->AddString("ListString2");
    pList1->AddString("ListString3");
    pList1->AddString("ListString4");
    pList1->AddString("ListString5");
    pList1->AddString("ListString6");
    pList1->SetCurSel(0);
}
```

```

        // Add strings to the combo box and select
        // the first string in the list.
        CComboBox* pCombo1 = (CComboBox*)GetDlgItem(IDC_COMBO1);
        pCombo1->AddString("ComboString1");
        pCombo1->AddString("ComboString2");
        pCombo1->AddString("ComboString3");
        pCombo1->AddString("ComboString4");
        pCombo1->AddString("ComboString5");
        pCombo1->AddString("ComboString6");
        pCombo1->SetCurSel(0);

        return TRUE;
    }

void CCnt1Dlg::OnOK()
{
    // Get the text in the edit box.
    CEdit* pEdit1 = (CEdit*)GetDlgItem(IDC_EDIT1);
    pEdit1->GetWindowText(m_edit1);

    // Get the states of the radio buttons.
    CButton* pRadio = (CButton*)GetDlgItem(IDC_RADIO1);
    m_radio1 = pRadio->GetCheck();
    pRadio = (CButton*)GetDlgItem(IDC_RADIO2);
    m_radio2 = pRadio->GetCheck();
    pRadio = (CButton*)GetDlgItem(IDC_RADIO3);
    m_radio3 = pRadio->GetCheck();

    // Get the states of the check boxes.
    CButton* pCheck = (CButton*)GetDlgItem(IDC_CHECK1);
    m_check1 = pCheck->GetCheck();
    pCheck = (CButton*)GetDlgItem(IDC_CHECK2);
    m_check2 = pCheck->GetCheck();
    pCheck = (CButton*)GetDlgItem(IDC_CHECK3);
    m_check3 = pCheck->GetCheck();

    // Get the list box's selected string.
    CListBox* pList1 = (CListBox*)GetDlgItem(IDC_LIST1);
    int listIndex = pList1->GetCurSel();
    pList1->GetText(listIndex, m_list1);

    // Get the combo box's selected string.
    CComboBox* pCombo1 = (CComboBox*)GetDlgItem(IDC_COMBO1);
    int comboIndex = pCombo1->GetCurSel();
    pCombo1->GetLBText(comboIndex, m_combo1);

    // Call the base class's OnOK() in order to
    // shut down the dialog box.
    CDialog::OnOK();
}

```

---

## Declaring a Friend Function

In Chapter 7, “Programming Dialog Boxes,” you may recall that I mentioned two ways for a window class to get access to a dialog box class’s data members. The first, which was demonstrated in that previous chapter, is to declare the dialog box class’s data members (at least, the data members that must participate in the data exchange) as public. While this solution is easy to accomplish, it’s not as elegant as it could be, because it grants access not only to the window class that must access the dialog box variables, but also to any other class that might try to gain such access.

- **See “Writing a Dialog Box Class,” p. 130**

A good way to limit this uncontrolled access is to make the window class that needs to read information from the dialog box class a friend of the dialog box class. A class declared as a friend of another class has access to that class’s public, protected, and even private data members. However, the access is limited to the friend class. No other class can access the protected and private data members.

The `Control1` application uses the friend access method to share the dialog box class’s variables with the main window class. If you look at the top of the `CCntDlg` class’s header file (see Listing 8.3), you see this line:

```
class CMainFrame;
```

This line tells the compiler that the identifier `CMainFrame` represents a class. Near the end of the `CCntDlg` class’s header file, you’ll see why the compiler needs this information. The line

```
friend CMainFrame;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb’s [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

tells the compiler to make the CMainFrame class a friend of the CCntIDlg class, giving CMainFrame access to all of CCntIDlg's data members. This is all the code it takes—just two lines—to limit outside access of CCntIDlg's variables to the CMainFrame class.

## Associating MFC Classes with Controls

As you know, MFC features many classes for window controls. You were introduced to these classes in Chapter 1, “An Introduction to MFC.” The control classes were also mentioned at the beginning of this chapter. Up until now, however, you haven't used these classes with the controls you added to your dialog boxes. This is because, in many cases, you don't need to manipulate a control at that level. You just display your dialog box and let the controls take care of themselves. There are times, though, when it's handy to be able to manipulate a control directly, which is what the many control classes such as CEdit, CButton, and CListBox enable you to do.

Each of the control classes features member functions that do everything from initialize the contents of the control to respond to Windows messages. But to use these member functions, you must first associate the control with the appropriate class. For example, to manipulate an edit box through MFC, you must first associate that control with the CEdit class. Then, you can use the CEdit class's member functions to manage the control.

To associate a control with a class, you must first get a pointer to the control. You can do this easily by calling the GetDlgItem() member function, which a dialog box class inherits from CWnd. You call GetDlgItem() like this:

```
CEdit* pEditControl = (CEdit*)GetDlgItem(IDC_EDIT1);
```

The GetDlgItem() function returns a pointer to a CWnd object. (Remember: Every control class has CWnd as a base class.) Its single argument is the resource ID of the control for which you want the pointer. To gain access to the member functions of a control class, the returned CWnd pointer must be cast to the appropriate type of pointer. In the previous line, you can see that GetDlgItem()'s return value is being cast to a CEdit pointer.

Once you have the pointer, you can access the class's member functions through that pointer. For example, to set the contents of the edit control for which the previous code line got a pointer, you'd use a line like this:

```
pEditControl->SetWindowText("Text for the edit control");
```

## Initializing the Dialog Box's Controls

In the Control1 application, the program uses the above technique to create a simple data transfer mechanism for the dialog box, without calling upon MFC's DDX functions. The

dialog box class first declares in its header file data members for holding the information the user entered into the dialog box, as shown in Listing 8.5.

#### **Listing 8.5 LST08\_05.CPP—Declaring Data Members for the Dialog Box Class**

---

```
protected:
    CString m_edit1;
    int m_radio1;
    int m_radio2;
    int m_radio3;
    int m_check1;
    int m_check2;
    int m_check3;
    CString m_list1;
    CString m_combol;
```

---

As you can see, there is one variable for each dialog box control with which the user can interact. Notice also that these data members are declared as protected. However, in spite of their protected status, the CMainFrame class can access them, because CMainFrame is a friend class of CCntDlg.

If you recall, the Control1 application's dialog box appears with default values already selected in its controls. For example, the edit box appears with the text "Default," and the first radio button in the radio button group is selected. Obviously, these controls are being initialized somewhere in the program—and that somewhere is the CCntDlg class's OnInitDialog() function. The OnInitDialog() function gets called as part of the dialog box creation process (specifically, it responds to the WM\_INITDIALOG Windows message). Because OnInitDialog() is a virtual function of the CDialog class, however, you don't need to create an entry in a message map. Just override the function in your custom dialog box class.

In your overridden OnInitDialog(), you must first call the base class's version, like this:

```
CDialog::OnInitDialog();
```

Then, you can perform whatever special initialization is required by your dialog box class. In the CCntDlg class, that initialization is setting the various controls to their default values. First, the program sets the edit box, like this:

```
CEdit* pEdit1 = (CEdit*)GetDlgItem(IDC_EDIT1);
pEdit1->SetWindowText("Default");
```

This code is very similar to the example you saw earlier in this section.

Next, the program sets the radio button group to its default state, which is the first button selected, like this:

```
CButton* pRadio1 = (CButton*)GetDlgItem(IDC_RADIO1);
pRadio1->SetCheck(TRUE);
```

The SetCheck() member function of the CButton class determines the check state of a

button. Although the preceding example uses TRUE as the function's argument, there are actually three possible settings: 0 turns off the check mark; 1 turns on the check mark; and 2 sets the button control to its "indeterminate" state. However, you can use 2 only when you've given the button the BS\_3STATE or BS\_AUTO3STATE style (which applies mostly to check boxes rather than radio buttons).

After setting the radio buttons, the program performs similar initialization on the first check box, like this:

```
CButton* pCheck1 = (CButton*)GetDlgItem(IDC_CHECK1);  
pCheck1->SetCheck(TRUE);
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Setting the checked state of buttons is pretty easy. Initializing a list box, however, takes a little extra work, as shown in Listing 8.6.

### Listing 8.6 LST08\_06.CPP—Initializing a List Box

```
CListBox* pList1 = (CListBox*)GetDlgItem(IDC_LIST1);
pList1->AddString("ListString1");
pList1->AddString("ListString2");
pList1->AddString("ListString3");
pList1->AddString("ListString4");
pList1->AddString("ListString5");
pList1->AddString("ListString6");
pList1->SetCurSel(0);
```

The AddString() member function of the CListBox class, adds a string to the contents of the list box. So, the code in Listing 8.6 adds the six selections that the user can choose from the list box. The SetCurSel() function, also a member of the CListBox class, determines which of the entries on the list box are selected. The function's single argument is the zero-based index of the item to select. So, an index of 0 selects the first item in the list. An index of -1 initializes the list box without a selection.

A combo box is not unlike a list box when it comes to initialization, as you can see in Listing 8.7.

### Listing 8.7 LST08\_07.CPP—Initializing a Combo Box

```
CComboBox* pCombo1 = (CComboBox*)GetDlgItem(IDC_COMBO1);
pCombo1->AddString("ComboString1");
pCombo1->AddString("ComboString2");
pCombo1->AddString("ComboString3");
pCombo1->AddString("ComboString4");
pCombo1->AddString("ComboString5");
pCombo1->AddString("ComboString6");
pCombo1->SetCurSel(0);
```

The preceding lines work exactly like the similar lines used to initialize the list box, adding six strings to the combo box's list, and then making the first string the default selection in the list. That is, the first string will already be entered into the

combo box's edit box when the dialog box appears.

---

**Tip:**

If you want to create a dialog box class that retains the most recently entered data, use `OnInitDialog()` to copy the contents of the appropriate class data members to the controls rather than using “hard-coded” data as is done in the `Control1` application. Using this method, you enable the Windows class that creates the dialog box object to initialize the dialog box's controls, by copying data into the dialog box class's data members before displaying the dialog box.

---

## Responding to the OK Button

Once the dialog box appears on the screen, the user can enter whatever data he likes into the dialog box's controls. The `Control1` application doesn't regain control until the user exits the dialog box, by clicking either the Cancel or OK buttons. As with most dialog boxes, if the user clicks the Cancel button (indicated by a return value of `IDCANCEL` from `DoModal()`), the program simply ignores any changes that may have been made in the dialog box. However, if the user exits the dialog box via the OK button, the program must transfer the dialog box's data from the controls to the dialog box class's data members. If the dialog box is allowed to close before the controls' contents are copied, the data disappears along with the dialog box.

So, in order to copy data from the dialog box's controls to the class's data members, a program must first know when the user has clicked the OK button. This is done by overriding the dialog box class's `OnOK()` member function. In `OnOK()`, the program associates controls with the appropriate control classes, and then uses the class member functions to perform whatever tasks are required to process the dialog box's data before the dialog box is deleted.

---

**Note:**

If your dialog box class needs to know—before the dialog box is removed from the screen—when the user has clicked the Cancel button, you can override the `OnCancel()` member function. After processing the Cancel button as required for your application, you'll usually want to call `CDialog::OnCancel()` to continue with the cancel process.

---

In the `Control1` application, the `OnOK()` function first extracts the contents of the edit box, like this:

```
CEdit* pEdit1 = (CEdit*)GetDlgItem(IDC_EDIT1);  
pEdit1->GetWindowText(m_edit1);
```

The first of these two lines gets a pointer to the edit control and casts it to a `CEdit` pointer. The program can then use the pointer to call the `CEdit` member function `GetWindowText()`, whose single argument is a string object into which the edit box's contents should be copied.

The program copies the contents of the radio buttons similarly, as shown in Listing 8.8.

## Listing 8.8 LST08\_08.CPP—Storing the Status of the Radio Buttons

---

```
// Get the states of the radio buttons.  
CButton* pRadio = (CButton*)GetDlgItem(IDC_RADIO1);  
m_radio1 = pRadio->GetCheck();  
pRadio = (CButton*)GetDlgItem(IDC_RADIO2);  
m_radio2 = pRadio->GetCheck();  
pRadio = (CButton*)GetDlgItem(IDC_RADIO3);  
m_radio3 = pRadio->GetCheck();
```

---

Here, after getting a pointer to each control and casting it to the CButton class, the program calls the CButton member function GetCheck() to obtain the status of each of the radio buttons. GetCheck() returns a 0, 1, or 2, depending on whether the control is in an unchecked, checked, or indeterminate state, respectively.

The program gets the contents of the check boxes in almost exactly the same way, as shown in Listing 8.9.

## Listing 8.9 LST08\_09.CPP—Storing the Status of the Check Boxes

---

```
CButton* pCheck = (CButton*)GetDlgItem(IDC_CHECK1);  
m_check1 = pCheck->GetCheck();  
pCheck = (CButton*)GetDlgItem(IDC_CHECK2);  
m_check2 = pCheck->GetCheck();  
pCheck = (CButton*)GetDlgItem(IDC_CHECK3);  
m_check3 = pCheck->GetCheck();
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

As you can see, the CButton class handles both radio buttons and check boxes. The only difference between Listing 8.8 and 8.9 is the resource IDs used to obtain pointers to the buttons.

The following lines show how the Control1 application extracts data from the list box control:

```
CListBox* pList1 = (CListBox*)GetDlgItem(IDC_LIST1);
int listIndex = pList1->GetCurSel();
pList1->GetText(listIndex, m_list1);
```

The first line above obtains a pointer to the list box control and associates it with the CListBox class. With the CListBox pointer in hand, the program can call the GetCurSel() member function, which returns the zero-based index of the selected item in the list box. Finally, a call to the CListBox member function GetText() copies the selected text string into the CCntIDlg class's m\_list1 data member. GetText()'s two arguments are the index of the text item to get and a reference to a CString object into which to store the string. (The second argument can also be a pointer to a char array.)

As you may have guessed, the program can copy the contents of the combo box in almost exactly the same manner, like this:

```
CComboBox* pCombo1 = (CComboBox*)GetDlgItem(IDC_COMBO1);
int comboIndex = pCombo1->GetCurSel();
pCombo1->GetLBText(comboIndex, m_combo1);
```

As always, the first line gets a pointer to the control; only this time the pointer is cast to a CComboBox pointer. The second line gets the zero-based index of the selected item in the combo box, whereas the third line copies the selected text line to the CCntIDlg class's m\_combo1 data member. The CComboBox class's GetLBText() member function works just like the CListBox class's GetText(), the two arguments being the index of the text item to get and a reference to a CString object into which to store the string. (Again, the second argument can also be a pointer to a char array.)

At this point in the program, all the important data has been copied from the dialog box's controls into the dialog box class's data members, which means it's now safe to remove the dialog box from the screen. Removing the dialog box is as simple as calling the base class's OnOK() member function, like this:

```
CDialog::OnOK();
```

---

**Note:**

You can also use the `OnOK()` member function to perform data validation. To do this, after extracting the contents of a control, determine whether the control's data is valid. If the data is not valid, display a message box to the user describing the problem, and then return from `OnOK()` without calling `CDialog::OnOK()`. Not calling `CDialog::OnOK()` leaves the dialog box on the screen so the user can correct the bad entry.

---

## Handling the Dialog Box in the Window Class

As you've seen, you can create your own data transfer and validation mechanisms by directly manipulating controls in a dialog box. Those DDX and DDV functions don't seem so mysterious now, do they? (You can, of course, do much more with your dialog box's controls than copy and validate data. In fact, once you have a pointer to a control, you can call any of the control's member functions.) The final step is to display your dialog box from within your main program, usually from a window class.

The `Control1` application displays the dialog box in response to its `Dialog`, `Test` command, which is handled by the `OnDialogTest()` message response function. In that function, the program first creates the dialog box object and then calls `DoModal()` to display it, like this:

```
CCntDlg dialog(this);
int result = dialog.DoModal();
```

If the user exits the dialog box by clicking the OK button, the function copies the dialog box's data into data members of the window class and then calls `Invalidate()` in order to repaint the window, which displays the current data from the dialog box. The code that handles these tasks is shown in Listing 8.10.

### Listing 8.10 LST08\_10.CPP—Copying the Dialog Box's Data

---

```
if (result == IDOK)
{
    // Save the contents of the dialog box.
    m_edit1 = dialog.m_edit1;
    m_radio1 = dialog.m_radio1;
    m_radio2 = dialog.m_radio2;
    m_radio3 = dialog.m_radio3;
    m_comb1 = dialog.m_comb1;
    m_check1 = dialog.m_check1;
    m_check2 = dialog.m_check2;
    m_check3 = dialog.m_check3;
    m_list1 = dialog.m_list1;

    // Force the window to repaint.
    Invalidate();
}
```

---

The dialog box object is automatically deleted when it goes out of scope, taking all its data with it, which is why you must copy whatever data you need from the dialog box.



[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## *Part II*

# *Programming Windows 95 Controls*

## Chapter 9

# Progress Bar, Slider, and Spinner Controls

- How to create the Windows 95 Controls App application
- About progress bar controls
- About slider controls
- About spinner controls

As a Windows user, you've long been accustomed to seeing controls like buttons, list boxes, menus, and edit boxes. As Windows developed, however, Microsoft noticed that applications developers routinely created other types of controls in their programs. These controls included things like toolbars, status bars, progress bars, tree views, and others. To make life easier for Windows programmers, when Microsoft created Windows 95 (and the latest version of Windows NT), it included these popular controls as part of the operating environment. Now, Windows programmers no longer need to create their own versions of these controls from scratch. In this chapter, you'll be introduced to several of Windows 95's common controls.

## The Win95 Controls Application



Over the course of the next few chapters, you will build a sample program called Win95 Controls App. You can find the program's executable file, as well as the complete source code, in the CHAP09 folder on this book's CD-ROM. When you run the program, you see the window shown in Figure 9.1. As you can see, Win95 Controls App demonstrates five of the Windows 95 common controls: the progress bar, slider, spinner, list view, and tree view controls. In this chapter, you'll learn the basics of creating and using three of these controls.

To get going quickly, you'll start creating the Win95 Controls application using AppWizard. Then you'll learn how to use MFC to create and manipulate the Windows 95 controls demonstrated by the application. To build the basic

Win95 Controls application, follow the steps given next.



The complete source code and executable file for this part of the Win95 application can be found in the CHAP09\Win95, Part 1 directory on this book's CD-ROM.

1. Choose Developer Studio's File, New command, and create a new AppWizard application called Win95, as shown in Figure 9.2.



**FIG. 9.1** The windows 95 Controls App demonstrates five of Windows 95's common controls.



**FIG. 9.2** Create a new AppWizard project workspace.

2. Choose the following options in the six MFC AppWizard pages. When the New Project Information dialog box appears, it should look like Figure 9.3.

Step 1	Single document.
Step 2 of 6	Use default options.
Step 3 of 6	Use default options.
Step 4 of 6	Turn off everything except 3D Controls.
Step 5 of 6	Use default options.
Step 6 of 6	Use default classes.

3. In the New Project Information dialog box, click the OK button. Developer Studio creates the new project's source code files.

4. Click the ResourceView tab to display the application's resources (see Figure 9.4).



**FIG. 9.3** When you've created the project, the New Project Information dialog box should look like this.



**FIG. 9.4** Click the ResourceView tab.

5. Double-click Win95 Resources in the ResourceView window, double-click Dialog, and then double-click IDD\_ABOUTBOX. The dialog box editor appears, as shown in Figure 9.5.

6. Use the editor to change the dialog box's title to **About Win95 Controls App**; to change the version line to **Win95 Controls App, Version 1.0**; and to add the line **by Macmillan Computer Publishing** to the dialog box below the copyright line, as shown in Figure 9.6.



**FIG. 9.5** When you click the dialog box resource's ID, the dialog box editor opens.



**FIG. 9.6** The finished dialog box should look like this.

7. Double-click Menu in the ResourceView menu and then double-click IDR\_MAINFRAME. The menu editor appears (see Figure 9.7).

8. Click the Edit menu (in the menu editor), and then press your keyboard's Delete key to remove the menu from the resource.

9. Click the File menu and then delete all the menu items except Exit.



**FIG. 9.7** When you double-click a menu resource ID, the menu editor appears.

10. Click the Help menu, and then double-click the About entry to bring up the Menu Items Properties dialog box. Change the About win95& menu item to **&amp;About Win95 Controls App&**.

11. Double-click String Table in the ResourceView window, and then double-click the String Table resource. The string table editor appears (see Figure 9.8).



**FIG. 9.8** When you double-click a string table resource, the string table editor appears.

**12.** Double-click the IDR\_MAINFRAME string in the string table. When the String Properties dialog box appears, change the first segment of the caption string from win95 to **Win95 Controls App** (see Figure 9.9). This is the string that appears as the application's title in the application's title bar.



**FIG. 9.9** Change the application's title in the IDR\_MAINFRAME string.

**13.** Double-click Accelerator in the ResourceView window, and then double-click IDR\_MAINFRAME. The accelerator editor appears (see Figure 9.10).



**FIG. 9.10** When you click an accelerator resource, the accelerator editor appears.

**14.** Use your keyboard's Delete key to delete all accelerators from the accelerator table. This prevents the user from accidentally selecting a hotkey for a command that the application does not support.

**15.** Load the application's MAINFRM.CPP file and then add the following lines to the beginning of the PreCreateWindow() function:

```
cs.cx = 480;
cs.cy = 440;
```

These lines set the size of the application's main window.

You have now completed the first part of the Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar; by selecting Developer Studio's Build, Build command; or by pressing F7. When you have the program compiled, you can run it if you like. However, all you'll see at this point is a blank window, as shown in Figure 9.11. In the following sections in this chapter, you'll add several Windows 95 common controls to the application, starting with the progress bar control. In the next chapter, you'll complete the application.



**FIG. 9.11** This is the basic Win95 Controls App, before any controls have been added.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE

BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## The Progress Bar Control

Probably the easiest to use of the new common controls is the *progress bar*, which is nothing more than a rectangle that fills in slowly with colored blocks. The more colored blocks filled in, the closer the task is to being complete. When the progress bar is completely filled in, the task associated with the progress bar is also complete. You might use a progress bar to show the status of a sorting operation or to give the user visual feedback about a large file that's being loaded.

To add the progress bar to the Win95 Controls App application, follow these steps.

The complete source code and executable file for this part of the Win95 application can be found in the CHAP09\Win95, Part 2 directory on this book's CD-ROM.



1. Use ClassWizard to add the OnCreate() function to the CWin95View class, as shown in Figure 9.12.
2. Click the Edit Code button and then add the following line to OnCreate(), right after the TODO: Add your specialized creation code here comment:

```
CreateProgressBar();
```

3. Add the function shown in Listing 9.1 to the end of the WIN95VIEW.CPP file.

### Listing 9.1 LST09\_01.CPP—The CreateProgressBar() Function

```
void CWin95View::CreateProgressBar()
{
    m_progressBar.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,
        CRect(20, 40, 250, 80), this, 102);
    m_progressBar.SetRange(1, 100);
    m_progressBar.SetStep(10);
    m_progressBar.SetPos(50);
    m_timer = FALSE;
}
```



**FIG. 9.12** Here's how to use ClassWizard to add the OnCreate() function to the application.

4. Add the following line to the OnDraw() function, right after the TODO: Add draw code

for native data here comment:

```
pDC->TextOut(20, 22, "Progress Bar Control");
```

**5.** Use ClassWizard to add the OnLButtonDown() function to the view class, as shown in Figure 9.13.



**FIG. 9.13** Here's how to use ClassWizard to add the OnLButtonDown() function to the application.

**6.** Click the Edit Code button and then add the lines shown in Listing 9.2 to OnLButtonDown(), right after the TODO: Add your message handler code here and/or call default comment.

#### **Listing 9.2 LST09\_02.CPP—Code for the Function**

---

```
if (m_timer)
{
    KillTimer(1);
    m_timer = FALSE;
}
else
{
    SetTimer(1, 500, NULL);
    m_timer = TRUE;
}
```

---

**7.** Use ClassWizard to add the OnTimer() function to the view class, as shown in Figure 9.14.



**FIG. 9.14** Here's how to use ClassWizard to add the OnTimer() function to the application.

**8.** Click the Edit Code button and then add the following line to OnTimer(), right after the TODO: Add your message handler code here and/or call default comment:

```
m_progressBar.StepIt();
```

**9.** Use ClassWizard to add the OnDestroy() function to the view class, as shown in Figure 9.15.

**10.** Click the Edit Code button and then add the following line to OnDestroy(), right after the TODO: Add your message handler code here comment:

```
KillTimer(1);
```



**11.** Load the WIN95VIEW.H file and then add the following lines to the class's Attribute section, right after the line `CWin95Doc* GetDocument()`:

```
protected:  
    CProgressCtrl m_progressBar;  
    BOOL m_timer;
```

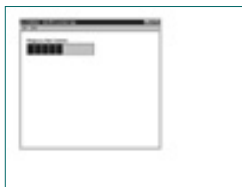


**FIG. 9.15** Here's how to use ClassWizard to add the `OnDestroy()` message response function to the application.

**12.** Add the following line to the class's Implementation section, right after the `protected` keyword:

```
void CreateProgressBar();
```

You have now completed the second part of the Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar; by selecting Developer Studio's **Build**, **Build** command; or by pressing F7. When you run the application now, you see the window shown in Figure 9.16.



**FIG. 9.16** The Win95 Controls App now sports a snazzy progress bar.

To see the progress bar in action, click anywhere in the background of Win95 Controls App's window. When you do, the progress bar starts filling with colored blocks. When the progress bar is completely filled, it starts over again. This continues until you click the window again or exit the program. Of course, in this program, the progress bar isn't tracking a real task in progress. It's simply responding to timer messages. However, the program still demonstrates how you might use a progress bar in your own applications.

## Creating the Progress Bar

This might be an obvious statement, but before you can use a progress bar, you must create it. Often in an MFC program, the controls are created as part of a dialog box. However, Win95 Controls App displays its controls in the application's main window. It does this by creating controls in the view class's `OnCreate()` function, which responds to the `WM_CREATE` Windows message. The progress bar control is declared as a data member of the view class, like this:

```
protected:  
CProgressCtrl m_progressBar;
```

As you can see, the progress bar is an object of the MFC `CProgressCtrl` class.

To create the progress bar control, `OnCreate()` calls the local `CreateProgressBar()` member function, like this:

```
CreateProgressBar( );
```

The fun begins in CreateProgressBar(). First, the function creates the progress bar control by calling the control's Create() function:

```
m_progressBar.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,  
    CRect(20, 40, 250, 80), this, 102);
```

This function's four arguments are the control's style flags, the control's size (as a CRect object), a pointer to the control's parent window, and the control's ID. (Usually, you declare a constant, such as IDC\_PROGRESSBAR, to use as the control's ID. I used a hard-coded value to keep the code as simple as possible.) The style constants are the same constants you would use for creating any type of window (a control is really nothing more than a special kind of window, after all). In this case, you need at least WS\_CHILD (which indicates that the control is a child window) and WS\_VISIBLE (which ensures that the user can see the control). The WS\_BORDER style is a nice addition, because it adds a dark border around the control, setting it off from the rest of the window.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 9.1 Member Functions of the CProgressCtrl Class

Function	Description
Create()	Creates the progress bar control
OffsetPos()	Advances the control the given number of blocks
SetPos()	Sets the control’s current value
SetRange()	Sets the control’s minimum and maximum values
SetStep()	Sets the value by which the control advances
StepIt()	Advances the control by one step unit

To initialize the control, you merely call the CProgressCtrl object’s appropriate member functions, which Win95 Controls App does like this:

```
m_progressBar.SetRange(1, 100);
m_progressBar.SetStep(10);
m_progressBar.SetPos(50);
```

The call to SetRange() determines the values represented by the progress bar. The two arguments are the minimum and maximum values. So, after the preceding call to SetRange(), if the progress bar is set to 1, it displays no colored blocks; in contrast, setting the control’s position to 100 fills the control with colored blocks.

Next, the CreateProgressBar() function calls SetStep(), which determines how far the progress bar advances with each increment. The larger this value, the faster the progress bar fills with colored blocks. Because the range and the step rate are related, a control with a range of 1–10 and a step rate of 1 works almost identically to a control with a range of 1–100 and a step rate of 10.

When the Win95 Controls App starts, the progress bar is already half filled with colored blocks. (This is purely for aesthetic reasons. Usually a progress bar begins its life empty.) This is because of the call to SetPos() with the value of 50, which is the midpoint of the control’s range.

Manipulating the Progress Bar

In Win95 Controls App, the progress bar starts counting forward when you click in the window’s background. This is because the program responds to the mouse click by starting a timer that sends WM\_TIMER messages to the program twice a second. In the view class’s OnTimer() function, the program makes the following function call:

```
m_progressBar.StepIt();
```

The StepIt() function increments the progress bar control’s value by the step rate, causing new blocks to be displayed in the control as the control’s value setting counts upward. When the control reaches its maximum, it automatically starts over.

---

### Note:

Notice that there are no CProgressCtrl member functions that control the size or number of blocks that will fit into the control. This attribute is controlled indirectly by the size of the control.

---

## The Slider Control

Many times in a program, the user might have enter a value that lies within a specific range. For this sort of task, you'd use MFC's CSliderCtrl class to create a *slider* (sometimes called a *trackbar*) control.

For example, suppose you need the user to enter a percentage that your program needs to calculate another value. In that case, you'd want the user to enter only values in the range from 0 to 100. Other values would be invalid and could cause problems in your program if such invalid values were not carefully trapped.

Using the slider control, you can force the user to enter a value in the specified range. Although the user can accidentally enter a wrong value (a value that doesn't accomplish what the user wants to do), he or she can't enter an invalid value (one that brings your program crashing down like a stone wall in an earthquake).

In the case of a percentage, you'd create a slider control with a minimum value of 0 and a maximum value of 100. Moreover, to make the control easier to position, you'd want to place tick marks at each setting that's a multiple of 10, giving 11 tick marks in all (including the one at 0). Win95 Controls App creates exactly this type of slider. The following steps show you how to add the control to Win95 Controls App.

<http://www.quecorp.com/semfc> The complete source code and executable file for this part of the Win95 application can be found in the CHAP09\Win95, Part 3 directory on this book's CD-ROM.



1. Load the WIN95VIEW.CPP file and then add the following line to the OnDraw() member function, right after the line pDC->TextOut(20, 22, "Progress BarControl") that you placed there previously:

```
pDC->TextOut(270, 22, "Slider Control:");
```

2. Add the following line to the OnCreate() function, right after the line CreateProgressBar() that you placed there previously:

```
CreateSlider();
```

3. Add the function shown in Listing 9.3 to the end of the WIN95VIEW.CPP file.

---

### Listing 9.3 LST09\_03.CPP—The CreateSlider() Function

---

```
void CWin95View::CreateSlider()
{
    m_slider.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |
        TBS_AUTOTICKS | TBS_BOTH | TBS_HORZ,
        CRect(270, 40, 450, 80), this, 101);
    m_slider.SetRange(0, 100, TRUE);
    m_slider.SetTicFreq(10);
}
```

```

        m_slider.SetLineSize(1);
        m_slider.SetPageSize(10);
    }

```

4. Use ClassWizard to add the OnHScroll() function to the view class, as shown in Figure 9.17.
5. Click the Edit Code button and then add the lines shown in Listing 9.4 to OnHScroll(), right after the // TODO: Add your message handler code here and/or call default comment.

---

#### Listing 9.4 LST09\_04.CPP—Code for the OnHScroll() Function

---

```

CSliderCtrl* slider = (CSliderCtrl*)pScrollBar;
int position = slider->GetPos();
    char s[10];
    wsprintf(s, "%d  ", position);
    CClientDC clientDC(this);
    clientDC.TextOut(390, 22, s);

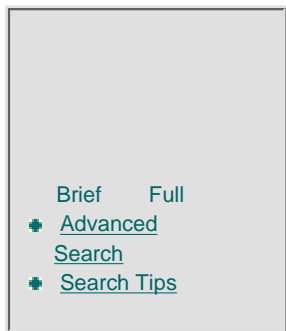
```

---



**FIG. 9.17** Here's how to use ClassWizard to add the OnHScroll() function to the program.

[Previous](#) [Table of Contents](#) [Next](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

### Initializing the Progress Bar

As soon as the progress bar control is created, it must be initialized. The CProgressCtrl class features a number of member functions that enable you to initialize and manipulate the control. Those member functions and their descriptions are listed in Table 9.1.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 9.2 Slider Styles

Style	Description
TBS_AUTOTICKS	Enables the slider to automatically draw its tick marks
TBS_BOTH	Draws tick marks on both sides of the slider
TBS_BOTTOM	Draws tick marks on the bottom of a horizontal slider
TBS_ENABLESELRANGE	Enables a slider to display a subrange of values
TBS_HORZ	Draws the slider horizontally
TBS_LEFT	Draws tick marks on the left side of a vertical slider
TBS_NOTICKS	Draws a slider with no tick marks
TBS_RIGHT	Draws tick marks on the right side of a vertical slider
TBS_TOP	Draws tick marks on the top of a horizontal slider
TBS_VERT	Draws a vertical slider

**SEARCH**  
 ITKNOWLEDGE

Brief    Full  
 + [Advanced Search](#)  
 + [Search Tips](#)

**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

6. Load the view class’s header file (WIN95VIEW.H) and then add the following line to the class’s Attributes section, right after the line BOOL m\_timer that you placed there previously:

```
CSliderCtrl m_slider;
```

7. Add the following line to the class’s Implementation section, right after the line void CreateProgressBar() that you placed there previously:

```
void CreateSlider();
```

You have now completed the third part of the Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar; by selecting Developer Studio’s Build, Build command; or by pressing F7. When you run the application now, you see the window shown in Figure 9.18.



**FIG. 9.18** The Win95 Controls App now has a slider control, as well as a progress bar control.

To see the slider work, click the slider's slot. When you do, the slider moves forward or backward, and the selected value appears to the right of the control's caption. As soon as the slider has the focus, you can also control it with your keyboard's Up Arrow and Down Arrow keys, as well as with the Page Up and Page Down keys. You can also drag the slider with your mouse to whatever position you like.

## Creating the Slider

In the Win95 the Controls App application, the slider is created in the `CreateSlider()` local member function, which, like `CreateProgressBar()`, the program calls from the view class's `OnCreate()` function. In `CreateSlider()`, the program first creates the slider control by calling its `Create()` member function, like this:

```
m_slider.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |  
                TBS_AUTOTICKS | TBS_BOTH | TBS_HORZ,  
                CRect(270, 40, 450, 80), this, 101);
```

This function's four arguments are the control's style flags, the control's size (as a `CRect` object), a pointer to the control's parent window, and the control's ID. The style constants include the same constants that you would use for creating any type of window, with the addition of special styles used with sliders. Table 9.2 lists these special styles.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**Table 9.3 Member Functions of the CSliderCtrl Class**

Function	Description
ClearSel()	Clears a selection from the control
ClearTics()	Clears tick marks from the control
Create()	Creates a slider control
GetChannelRect()	Gets the size of the control's slider
GetLineSize()	Gets the control's line size
GetNumTics()	Gets the number of tick marks
GetPageSize()	Gets the control's page size
GetPos()	Gets the control's position
GetRange()	Gets the control's minimum and maximum values
GetRangeMax()	Gets the control's maximum value
GetRangeMin()	Gets the control's minimum value
GetSelection()	Gets the current range selection
GetThumbRect()	Gets the size of the control's thumb
GetTic()	Gets the position of a tick mark
GetTicArray()	Gets all the control's tick positions
GetTicPos()	Gets the client coordinates of a tick mark
SetLineSize()	Sets the control's line size
SetPageSize()	Sets the control's page size
SetPos()	Sets the control's position
SetRange()	Sets the control's minimum and maximum values
SetRangeMax()	Sets the control's maximum value
SetRangeMin()	Sets the control's minimum value
SetSelection()	Sets a selected subrange in the control
SetTic()	Sets the position of a tick mark
SetTicFreq()	Sets the control's tick frequency
VerifyPos()	Determines whether the control's position is valid

Usually, when you create a slider control, you'll want to set the control's range and tick frequency. If the user is going to use the control from his or her keyboard, you also need to set the control's line and page size. In the Win95 Controls App application, the program initializes the slider as shown in Listing 9.5.

#### **Listing 9.5 LST09\_05.CPP—Initializing the Slider Control**

```
m_trackbar.SetRange(0, 100, TRUE);
m_trackbar.SetTicFreq(10);
m_trackbar.SetLineSize(1);
m_trackbar.SetPageSize(10);
```

---

The call to `SetRange()` sets the slider's minimum and maximum values to 0 and 100, respectively. The arguments are the minimum value, the maximum value, and a Boolean value indicating whether the slider should redraw itself after setting the range. Next, the call to `SetTicFreq()` ensures that there will be a tick mark at each interval of 10. (Without this function call, the slider would have a tick mark for each possible setting, 101 in all.) Finally, the call to `SetLineSize()` determines how much the slider moves when the user presses his or her Up Arrow or Down Arrow keys, and the call to `SetPageSize()` determines how much the slider moves when the user presses the Page Up or Page Down keys.

## Manipulating the Slider

When you get down to it, a slider is really a special scroll bar control. As such, when the user moves the slider, the control generates `WM_HSCROLL` messages, which Win95 Controls App captures in its view class's `OnHScroll()` member function, as shown in Listing 9.6.

### Listing 9.6 LST09\_06.CPP—Responding to a Slider Control

---

```
void CWin95View::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default

    CSliderCtrl* slider = (CSliderCtrl*)pScrollBar;
    int position = slider->GetPos();
    char s[10];
    wsprintf(s, "%d", position);
    CClientDC clientDC(this);
    clientDC.TextOut(390, 22, s);

    CView::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

---

`OnHScroll()`'s fourth parameter is a pointer to the scroll object that generated the `WM_HSCROLL` message. The preceding function first casts this pointer to the `CSliderCtrl` pointer. It then gets the current position of the slider by calling the `CSliderCtrl` member function `GetPos()`. As soon as the program has the slider's position, it converts the integer to a string and displays that string in the window.

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Initializing the Slider

As soon as the slider control is created, it must be initialized. The CSliderCtrl class features many member functions that enable you to initialize and manipulate the control. Those member functions and their descriptions are listed in Table 9.3.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## The Spinner Control

The slider control isn't the only way you can get a value in a predetermined range from the user. If you don't care about using the slider for visual feedback, you can use a ***spinner control***, which is little more than a couple of arrows that the user clicks to raise or lower the control's setting. If the slider control is a scroller with only a bar and a thumb, then a spinner control is the leftover arrow buttons. Follow the steps that come next to add a spinner control to Win95 Controls App.

The complete source code and executable file for this part of the Win95 application can be found in the CHAP09\Win95, Part 4 directory on this book's CD-ROM.



1. Load the WIN95VIEW.CPP file and then add the following line to the OnDraw() member function, right after the line pDC->TextOut(270, 22, "Slider Control:"), which you placed there previously:

```
pDC->TextOut(20, 102, "Spinner Control");
```

2. Add the following line to the OnCreate() function, right after the line CreateSlider(), which you placed there previously:

```
CreateSpinner();
```

3. Add the function shown in Listing 9.7 to the end of the WIN95VIEW.CPP file.

### Listing 9.7 LST09\_07.CPP—The CreateSpinner() Function

```
void CWin95View::CreateSpinner()
{
    m_buddyEdit.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,
        CRect(50, 120, 110, 160), this, 103);
    m_spinner.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |
        UDS_ALIGNRIGHT | UDS_SETBUDDYINT | UDS_ARROWKEYS,
        CRect(0, 0, 0, 0), this, 104);
    m_spinner.SetBuddy(&m_buddyEdit);
    m_spinner.SetRange(1, 100);
    m_spinner.SetPos(50);
}
```

4. Load the view class's header file (WIN95VIEW.H) and then add the following lines to the class's Attributes section, right after the line CSliderCtrl m\_slider that you placed there previously:

```
CSpinButtonCtrl m_spinner;  
CEdit m_buddyEdit;
```

5. Add the following line to the class's Implementation section, right after the line `void CreateSlider()` that you placed there previously:

```
void CreateSpinner();
```

You have now completed the fourth part of the Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar; by selecting Developer Studio's Build, Build command; or by pressing F7. When you run the application now, you see the window shown in Figure 9.19.



**FIG. 9.19** This version of Win95 Controls App adds a spinner to the other controls.

In the Win95 Controls App application, you can change the setting of the spinner control by clicking either of its arrows. When you do, the value in the attached edit box changes, indicating the spinner control's current setting. As soon as the control has the focus, you can also change its value by pressing your keyboard's Up Arrow and Down Arrow keys.

## Creating the Spinner Control

In the Win95 Controls App application, the spinner control is created in the `CreateSpinner()` local member function, which the program calls from the view class's `OnCreate()` function. In `CreateSpinner()`, the program creates the spinner control by first creating the associated buddy control to which the spinner control communicates its current value. In this case, as is typical, the buddy control is an edit box, which is created by calling the `CEdit` class's `Create()` member function:

```
m_buddyEdit.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,  
    CRect(50, 120, 110, 160), this, 103);
```

This function's four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. As you might remember from the control declarations, `m_buddyEdit` is an object of the `CEdit` class.

Now that the program has created the buddy control, it can create the spinner control in much the same way, by calling the object's `Create()` member function, like this:

```
m_spinner.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |  
    UDS_ALIGNRIGHT | UDS_SETBUDDYINT | UDS_ARROWKEYS,  
    CRect(0, 0, 0, 0), this, 104);
```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 9.4 Spinner Styles

Style	Description
UDS_ALIGNLEFT	Places the spinner control on the left edge of the buddy control
UDS_ALIGNRIGHT	Places the spinner control on the right edge of the buddy control
UDS_ARROWKEYS	Enables the user to change the control’s values using the keyboard’s Up Arrow and Down Arrow keys
UDS_AUTOBUDDY	Makes the previous window the buddy control
UDS_HORZ	Creates a horizontal spinner control
UDS_NOTHOUSANDS	Eliminates separators between each set of three digits
UDS_SETBUDDYINT	Displays the control’s value in the buddy control
UDS_WRAP	Causes the control’s value to wrap around to its minimum when the maximum is reached and vice versa

Initializing the Spinner Control

As soon as the spinner control is created, it must be initialized. The CSpinButtonCtrl class features member functions that enable you to initialize and manipulate the control. Those member functions and their descriptions are listed in Table 9.5.

Table 9.5 CSpinButtonCtrl Member Functions

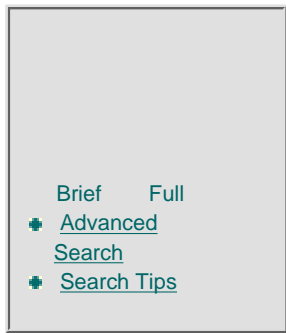
Function	Description
Create()	Creates the spinner control
GetAccel()	Gets the control’s speed
GetBase()	Gets the control’s numerical base
GetBuddy()	Gets a pointer to the control’s buddy control
GetPos()	Gets the control’s position
GetRange()	Gets the control’s minimum and maximum values
SetAccel()	Sets the control’s speed
SetBase()	Sets the control’s numerical base (10 for decimal, 16 for hex)
SetBuddy()	Sets the control’s buddy control
SetPos()	Sets the control’s position
SetRange()	Sets the control’s minimum and maximum values

After creating a spinner control, you’ll usually want to set the control’s buddy, range, and position. In the Win95 Controls App application, the program initializes the control like this:

```
m_spinner.SetBuddy(&mpm_buddyEdit);
m_spinner.SetRange(1, 100);
m_spinner.SetPos(50);
```

Here, the spinner control's buddy is set to the edit box that was first created in `CreateSpinner()`. Then the program calls `SetRange()` and `SetPos()` to give the control its starting range and position, respectively. Thanks to the `UDS_SETBUDDYINT` flag passed to `Create()` and the call to the control's `SetBuddy()` member function, Win95 Controls App needs to do nothing else to have the control's value appear on the screen. The control handles its buddy automatically.

[Previous](#) [Table of Contents](#) [Next](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

As you can guess by now, this function's four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. As with most controls, the style constants include the same constants that you would use for creating any type of window. However, the `CSpinButtonCtrl` class, of which `m_spinner` is an object, defines special styles to be used with spinner controls. Table 9.4 lists these special styles.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

## Chapter10

# Image List, List View, and Tree View Controls

- How to create image list controls
- How to create list view controls
- How to add columns and items to a list view control
- About creating tree view controls
- How to add items to tree view controls
- About notification messages

Two of the most complex controls to use in your programs are the list view and tree view controls. To add to the confusion, both of these controls must be associated with image list controls that contain the icons that will be displayed in the control. The image list control, although basically little more than an array of pictures, is one of the most misunderstood controls, probably because it doesn't appear on the screen, but rather stays hidden in memory like other types of data structures. In this chapter, you'll get a chance to create and use all three of these special controls.

## The Image List Control

Often in programs, you need to use a lot of images that are related in some way. For example, your application might have a toolbar with many command buttons, each of which uses a bitmap for its icon. In a case like this, it would be great to have some sort of program object that could not only hold the bitmaps, but also organize them so they can be accessed easily. That's exactly what an image list control does for you. An image list does nothing more than store a list of related images. You can use the images any way you see fit in your program. However, several Windows 95 controls rely on or use image lists. These controls are:

- List view controls
- Tree view controls
- Property pages
- Toolbars

Besides the preceding controls, you will undoubtedly come up with many other uses for image lists. You might, for example, have an animation sequence that you'd like to display in a window. An image list is the perfect storage place for the frames that

make up the animation because you can easily access any frame just by using an index.

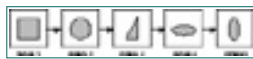
---

**Note:**

If the word *index* makes you think of arrays, you're close to understanding how an image list stores images. An image list is much like an array that holds pictures rather than integers or floating-point numbers. Just as with an array, you initialize each *element* of an image list and thereafter can access any part of the *array* using an index.

---

You won't, however, ever see in your running application an image list control in the same way that you can see a status bar or a progress bar control. That's because (again, like an array) an image list is nothing more than a storage structure for pictures. You can display the images stored in an image list, but you can't display the image list itself. Figure 10.1 shows how an image list is organized.



**FIG. 10.1** An image list is much like an array of pictures.

Now that you have some idea of what an image list is, you can add one to Win95 Controls App. Because image list controls are so closely related to list view and tree view controls, you'll create the application's image lists in the functions that will eventually create the application's list view and tree view controls. Complete the steps that follow to perform these tasks.



The complete source code and executable file for this part of the Win95 application can be found in the CHAP10\Win95, Part 5 directory of this book's CD-ROM.

1. Load the WIN95VIEW.CPP file, and add the following lines to the OnCreate() function, right after the line CreateSpinner(), which you placed there previously.

```
CreateListView();  
CreateTreeView();
```

2. Add the functions shown in Listing 10.1 to the end of the WIN95VIEW.CPP file.

---

**Listing 10.1 LST10\_01.CPP—The CreateListView() and CreateTreeView() functions**

---

```
void CWin95View::CreateListView()  
{  
    // Create the Image List controls.  
    m_smallImageList.Create(16, 16, FALSE, 1, 0);  
    m_largeImageList.Create(32, 32, FALSE, 1, 0);  
    HICON hIcon = ::LoadIcon (AfxGetResourceHandle(),  
        MAKEINTRESOURCE(IDI_ICON1));  
    m_smallImageList.Add(hIcon);  
    hIcon = ::LoadIcon (AfxGetResourceHandle(),
```

```

        MAKEINTRESOURCE(IDI_ICON2));
m_largeImageList.Add(hIcon);
}

void CWin95View::CreateTreeView()
{
    // Create the Image List.
    m_treeImageList.Create(13, 13, FALSE, 3, 0);
    HICON hIcon = ::LoadIcon(AfxGetResourceHandle(),
        MAKEINTRESOURCE(IDI_ICON3));
    m_treeImageList.Add(hIcon);
    hIcon = ::LoadIcon(AfxGetResourceHandle(),
        MAKEINTRESOURCE(IDI_ICON4));
    m_treeImageList.Add(hIcon);
    hIcon = ::LoadIcon(AfxGetResourceHandle(),
        MAKEINTRESOURCE(IDI_ICON5));
    m_treeImageList.Add(hIcon);
}

```

3. Copy the ENVELOP1.ICO, ENVELOP2.ICO, ICON3.ICO, ICON4.ICO, and ICON5.ICO files from this book's CD-ROM to Win95 Controls App's RES directory.
4. Select Developer Studio's Insert, Resource command. The Insert Resource dialog box appears, as shown in Figure 10.2.
5. Click the Import button, and then use the Import Resource file browser (see Figure 10.3) to select the five icons you copied in step 4. The ENVELOP1.ICO file should get the resource ID IDI\_ICON1, and the ENVELOP2.ICO file should get the ID IDI\_ICON2. The files ICON3.ICO, ICON4.ICO, and ICON5.ICO should get the IDs IDI\_ICON3, IDI\_ICON4, and IDI\_ICON5, respectively. Click the Import button to finalize your choices.



**FIG. 10.2** You use the Insert Resource dialog box to add resources to your project.

---

**Tip:**

To be sure that Developer Studio assigns the right IDs to the icons, you should import them one at a time, in the order listed in step 5. If you choose to import the icons all at once, you might have to change some of the assigned IDs to make them match the required IDs given in this step.

---



**FIG. 10.3** Use the Import Resource file browser to select the icon files.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 10.1 Member Functions of the CImageList Class**

Function	Description
Add()	Adds an image to the image list
Attach()	Attaches an existing image list to an object of the CImageList class
BeginDrag()	Starts an image-dragging operation
Create()	Creates an image list control
DeleteImageList()	Deletes an image list
Detach()	Detaches an image list from an object of the CImageList class
DragEnter()	Locks a window for updates and shows the drag image
DragLeave()	Unlocks a window for updates
DragMove()	Moves the drag image
DragShowNolock()	Handles the drag image without locking the window
Draw()	Draws an image that's being dragged
EndDrag()	Ends an image-dragging operation
ExtractIcon()	Creates an icon from an image
GetBkColor()	Gets an image list's background color
GetDragImage()	Gets the image for drag operations
GetImageCount()	Gets the number of images in the control
GetImageInfo()	Gets image information
GetSafeHandle()	Gets an image list's handle
Read()	Gets an image list from the given archive
Remove()	Removes an image from the image list
Replace()	Replaces one image with another
SetBkColor()	Sets an image list's background color
SetDragCursorImage()	Creates an image for drag operations
SetOverlayImage()	Sets the index of an overlay mask
Write()	Writes an image list to the given archive

[Brief](#) [Full](#)[Advanced Search](#)  
[Search Tips](#)

6. Load the view class's header file (WIN95VIEW.H), and add the following lines to the class's Attributes section, right after the line CEdit m\_buddyEdit, which you placed there previously.

```
CImageList m_smallImageList;  
CImageList m_largeImageList;  
CImageList m_treeImageList;
```

7. Add the following lines to the class's Implementation section, right after the line void CreateSpinner() you placed there previously.

```
void CreateListView();  
void CreateTreeView();
```

---

**Tip:**

To be sure that Developer Studio assigns the right IDs to the icons, you should import them one at a time, in the order listed in step 5. If you choose to import the icons all at once, you might have to change some of the assigned IDs to make them match the required IDs given in this step.

---

You have now completed the fifth part of Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar, by selecting Developer Studio's Build, Build command, or by pressing F7. When you run the application, it won't look any different from the previous version. However, now the application creates three image list controls that it will use with the list view and tree view controls you create later in this chapter.

## Creating the Image List

In the Win95 Controls App application, image lists are used with the list view and tree view controls, so the image lists for the controls are created in the CreateListView() and CreateTreeView() local member functions, which the program calls from the view class's OnCreate() function. You create an image list, which is an object of the CImageList class, like this:

```
m_smallImageList.Create(16, 16, FALSE, 1, 0);
```

The Create() function's five arguments are the width of the pictures in the control, the height of the pictures, a Boolean value indicating whether the images contain a mask, the number of images initially in the list, and the number of images by which the list can dynamically grow. This last value is 0 to indicate that the list is not allowed to grow at runtime. The CImageList class overloads the Create() function so that you can create image lists in various ways. You can find the other versions of Create() in your Visual C++ online documentation.

## Initializing the Image List

After you have an image list created, you'll want to add images to it. After all, an empty image list isn't of much use. The easiest way to add the images is to have the images as part of your application's resource file and to load them from there. For example, the following code shows how Win95 Controls App loads an icon into one of its image lists.

```
HICON hIcon = ::LoadIcon (AfxGetResourceHandle(),  
    MAKEINTRESOURCE( IDI_ICON1 ) );  
m_smallImageList.Add(hIcon);
```

Here, the program first gets a handle to the icon. Then, it adds the icon to the image list by calling the image list's Add() member function. Table 10.1 lists other member functions you can use to manipulate an object of the CImageList class. As you can see, you have a lot of control over an image list if you really want to dig in. In the following sections, you'll see how to use an image list with the list view and tree view controls.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief
 Full

• [Advanced Search](#)

• [Search Tips](#)



[an error occurred while processing this directive]

## The List View Control

Often, computer programs need to work with lists of objects and to organize those objects in such a way that the program’s user can easily determine each object’s attributes. An example is a group of files on a disk. Each file is a separate object that is associated with a number of attributes including the file’s name, size, and the date the file was last modified. Windows shows files either as icons in a window or as a table of entries, each entry showing the attributes associated with the files. The user has full control over the way the file objects are displayed, including which attributes are shown and which are not listed. The Windows 95 common controls include something called a list view control, which enables Windows 95 programmers to organize lists in exactly the same way Windows 95 does with files and other objects.

If you’d like to see an example of a full-fledged list view control, just open the Windows 95 Explorer (see Figure 10.4). The right side of the window shows how the list view control can organize objects in a window. (The left side of the window contains a tree view control, which you learn about later in this chapter, in the section titled “The Tree View Control.”) In the figure, the list view is currently set to the report view, in which each object in the list gets its own line showing, not only the object’s name, but also the attributes associated with that object.



**FIG. 10.4** Explorer uses list view controls to organize file information.

As I mentioned previously in this chapter, the user can change the way objects are organized in a list view control. Figure 10.5, for example, shows the list view portion of the Explorer set to the large icon setting, whereas Figure 10.6 shows the small icon setting, which enables the user to see more objects (in this case, files) in the window. The list view control also provides the user with the ability to edit the names of objects in the list, as well as to sort objects based on data displayed in a particular column. (This latter function works only when the list view control is in report view.)

**Tip:**

To set Explorer’s list view control to its different views, use the Large Icon, Small Icon, List, and Details commands on Explorer’s View menu. If you have Explorer’s toolbar displayed, you can also select these commands by clicking the appropriate toolbar buttons.



**FIG. 10.5** Here’s Explorer’s list view control set to large icons.

The Win95 Controls App application also sports a list view control, although it’s not as fancy as Explorer’s. To add the control to the application, complete the steps that follow.





```

m_listView.SetItemText(1, 1, "Sub Item 1.1");
m_listView.SetItemText(1, 2, "Sub Item 1.2");
lvItem.iItem = 2;
lvItem.iSubItem = 0;
lvItem.pszText = "Item 2";
m_listView.InsertItem(&amlvItem);
m_listView.SetItemText(2, 1, "Sub Item 2.1");
m_listView.SetItemText(2, 2, "Sub Item 2.2");

// Create the view-control buttons.
m_smallButton.Create("Small", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 120, 450, 140), this, 106);
m_largeButton.Create("Large", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 145, 450, 165), this, 107);
m_listButton.Create("List", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 170, 450, 190), this, 108);
    m_reportButton.Create("Report", WS_VISIBLE | WS_CHILD | WS_BORDER,
        CRect(400, 195, 450, 215), this, 109);

```

3. Add the lines shown in Listing 10.3 to the end of the view class's message map, found near the top of the WIN95VIEW.CPP file. Add the lines right after the line `//}}AFX_MSG_MAP`, which is already in the message map.

---

#### **Listing 10.3 LST10\_03.CPP—New Message Map Lines**

---

```

ON_COMMAND(106, OnSmall)
ON_COMMAND(107, OnLarge)
ON_COMMAND(108, OnList)
ON_COMMAND(109, OnReport)

```

4. Add the functions shown in Listing 10.4 to the end of the WIN95VIEW.CPP file.

---

#### **Listing 10.4 LST10\_04.CPP—New Functions for the Win95 ControlsApplication**

---

```

void CWin95View::OnSmall()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
        WS_VISIBLE | WS_CHILD | WS_BORDER |
        LVS_SMALLICON | LVS_EDITLABELS);
}

void CWin95View::OnLarge()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
        WS_VISIBLE | WS_CHILD | WS_BORDER |
        LVS_ICON | LVS_EDITLABELS);
}

void CWin95View::OnList()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
        WS_VISIBLE | WS_CHILD | WS_BORDER |
        LVS_LIST | LVS_EDITLABELS);
}

void CWin95View::OnReport()
{

```

```
SetWindowLong(m_listView.m_hWnd, GWL_STYLE,  
    WS_VISIBLE | WS_CHILD | WS_BORDER |  
    LVS_REPORT | LVS_EDITLABELS);  
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 10.2 List View Styles**

Style	Description
LVS_ALIGNLEFT	Left aligns items in the large icon and small icon views
LVS_ALIGNTOP	Top aligns items in the large icon and small icon views
LVS_AUTOARRANGE	Automatically arranges items in the large icon and small icon views
LVS_EDITLABELS	Enables the user to edit item labels
LVS_ICON	Sets the control to the large icon view
LVS_LIST	Sets the control to the list view
LVS_NOCOLUMNHEADER	Shows no column headers in report view
LVS_NOITEMDATA	Stores only the state of each item
LVS_NOLABELWRAP	Disallows multiple-line item labels
LVS_NOSCROLL	Turns off scrolling
LVS NOSORTHEADER	Turns off the button appearance of column headers
LVS_OWNERDRAWFIXED	Enables owner-drawn items in report view
LVS_REPORT	Sets the control to the report view
LVS_SHAREIMAGELISTS	Prevents the control from destroying its image lists when the control no longer needs them
LVS_SINGLESEL	Disallows multiple selection of items
LVS_SMALLICON	Sets the control to the small icon view
LVS_SORTASCENDING	Sorts items in ascending order
LVS_SORTDESCENDING	Sorts items in descending order

5. Use ClassWizard to add the OnNotify() function to the view class, as shown in Figure 10.7.

6. Click the Edit Code button, and add the lines shown in Listing 10.5 to the OnNotify() function. Place the code right after the TODO: Add your specialized code here and/or call the base class comment.



**FIG. 10.7** Use ClassWizard to add OnNotify() to the application.

#### **Listing 10.5 LST10\_05.CPP—Lines for the OnNotify() Function**

```
LV_DISPINFO* lv_dispInfo = (LV_DISPINFO*) lParam;

if (lv_dispInfo->hdr.code == LVN_BEGINLABELEDIT)
{
    CEdit* pEdit = m_listView.GetEditControl();
    // Manipulate edit control here.
}
else if (lv_dispInfo->hdr.code == LVN_ENDLABELEDIT)
{
    if ((lv_dispInfo->item.pszText != NULL) &&
        (lv_dispInfo->item.iItem != -1))
    {
        m_listView.SetItemText(lv_dispInfo->item.iItem,
                                0, lv_dispInfo->item.pszText);
    }
}
```

7. Load the view class's header file (WIN95VIEW.H), and add the lines shown in Listing 10.6 to the class's Attributes section, right after the line CImageList m\_treeImageList, which you placed there previously.

#### **Listing 10.6 LST10\_06.CPP—New Data Member Declarations**

```
CListCtrl m_listView;
CButton m_smallButton;
CButton m_largeButton;
CButton m_listButton;
CButton m_reportButton;
```

8. Add the lines shown in Listing 10.7 to the class's message map-function section, right after the line `//}}AFX_MSG`, which is already there.

---

**Listing 10.7 LST10\_07.CPP—Message Map Function Declarations**

---

```
afx_msg void OnSmall();  
afx_msg void OnLarge();  
afx_msg void OnList();  
afx_msg void OnReport();
```

---

You have now completed the sixth part of the Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar, by selecting Developer Studio's Build, Build command, or by pressing F7. When you run the application now, you see the window shown in Figure 10.8.

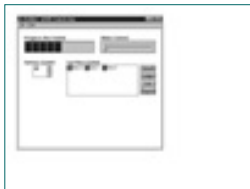


**FIG. 10.8** Here is Win95 Controls App with its new list view control.

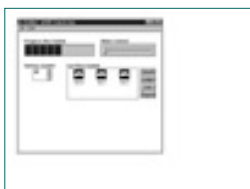
To switch between the small icon, large icon, list, and report views, click the appropriate button to the right of the control. Figure 10.9 shows the application's list view control displaying small icons, whereas Figure 10.10 shows the large icons.

### Creating the List View

In Win95 Controls App, the list view control is created in the `CreateListView()` local member function, which the program calls from the view class's `OnCreate()` function. In `CreateListView()`, the program first creates two image lists, as you saw in the previous section on image lists and as is shown in Listing 10.8. One image list will hold the small icon for the list view, and the other will hold the large icon. In this case, each list includes only one icon.



**FIG. 10.9** Here's the list view control set to small icons.



**FIG. 10.10** Here's the list view control set to large icons.

---

**Listing 10.8 LST10\_08.CPP—Creating the List View Control's Image Lists**

---

```
m_smallImageList.Create(16, 16, FALSE, 1, 0);  
m_largeImageList.Create(32, 32, FALSE, 1, 0);  
HICON hIcon = ::LoadIcon (AfxGetResourceHandle(),  
    MAKEINTRESOURCE(IDI_ICON1));  
m_smallImageList.Add(hIcon);  
hIcon = ::LoadIcon (AfxGetResourceHandle(),  
    MAKEINTRESOURCE(IDI_ICON2));  
m_largeImageList.Add(hIcon);
```

---

Now that the program has created the image lists, it can create the list view control, by calling the class's `Create()` member function, like this:

```
m_listView.Create(WS_VISIBLE | WS_CHILD | WS_BORDER |  
    LVS_REPORT | LVS_NOSORTHEADER | LVS_EDITLABELS,  
    CRect(160, 120, 394, 220), this, 105);
```

Here, `Create()`'s four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. The `CListCtrl` class, of which `m_listView` is an object, defines special styles to be used with list view controls. Table 10.2 lists these special styles and their descriptions.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)

[Brief](#)
[Full](#)

[Advanced Search](#)
[Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

## Initializing the List View

Although not especially difficult, setting up a list view control is quite a bit more work than setting up a simpler control like a progress bar. As you already know, the list view control uses two image lists, one for its small icons and one for its large icons. A list view control also uses column headers for its report view, as well as list items and subitems. In short, to initialize a list view control, you must complete the following steps:

1. Create the list view control.
2. Associate the control with its image lists.
3. Create a column object for each column that will appear in the report view.
4. Create list items and subitems for each item that will be displayed in the list view control.

In the following sections, you'll learn more about how to implement the steps in the previous list.

## Associating the List View with Its Image Lists

You've already created your list view control, so step one is out of the way. Now, you must associate the control with its image lists (which have also already been created). Win95 Controls App App handles the task like this:

```
m_listView.SetImageList(&amp;m_smallImageList, LVSIL_SMALL);
m_listView.SetImageList(&amp;m_largeImageList, LVSIL_NORMAL);
```

As you can see, the SetImageList() member function takes two parameters, which are a pointer to the image list and a flag indicating how the list is to be used. The SetImageList() function returns a pointer to the previously set image list, if any.

---

### Note:

There are three constants defined for SetImageList()'s second argument: LVSIL\_SMALL (which indicates that the list contains small icons), LVSIL\_NORMAL (large icons), and LVSIL\_STATE (state images).

---

## Creating the List View's Columns

The next task is to create the columns for the control's report view. You need one main column for the item itself and one column for each subitem associated with an item. For example, in Explorer's list view, the main column holds file and folder names. Each additional column holds the subitems for each item, including the file's size, type, and modification date. To create a column, you must first declare an LV\_COLUMN structure. You use this structure to pass information to and from the system. The LV\_COLUMN structure is defined as shown in Listing 10.9.



## Listing 10.9 LST10\_09.CPP—The LV\_COLUMN Structure

---

```
typedef struct _LV_COLUMN
{
    UINT mask;           // Flags indicating valid fields
    int fmt;             // Column alignment
    int cx;              // Column width
    LPSTR pszText;       // Address of string buffer
    int cchTextMax;      // Size of the buffer
    int iSubItem;        // Subitem index for this column
} LV_COLUMN;
```

---

The mask member of the structure must be set so that it indicates which of the other fields in the structure are valid. The flags you can use are LVCF\_FMT (meaning fmt is valid), LVCF\_SUBITEM (iSubItem is valid), LVCF\_TEXT (pszText is valid), and LVCF\_WIDTH (cx is valid). Essentially, when you give mask its value, you're telling the system which members of the structure to use and which to ignore.

The fmt member gives the column's alignment and can be LVCFMT\_CENTER, LVCFMT\_LEFT, or LVCFMT\_RIGHT. The alignment determines how the column's label and items are positioned in the column.

---

### Note:

The first column, which contains the main items, is always aligned to the left. The other columns in the report view can be aligned however you like.

---

The cx field specifies the width of each column, whereas pszText is the address of a string buffer. When you're using the structure to create a column (you also can use this structure to obtain information about a column), this string buffer contains the column's label. The cchTextMax member gives the size of the string buffer and is valid only when retrieving information about a column.

In Win95 Controls App's CreateListView() function, the program starts initializing the LV\_COLUMN structure as shown in Listing 10.10.

## Listing 10.10 LST10\_10.CPP—Initializing the LV\_COLUMN Structure

---

```
LV_COLUMN lvColumn;
lvColumn.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
lvColumn.fmt = LVCFMT_CENTER;
lvColumn.cx = 75;
```

---

The values being set in Listing 10.10 will be the same for every column created, so they are done first and then are not changed as the columns are created.

Next, the program creates its main column by setting the appropriate structure members and then calling the CListCtrl object's InsertColumn() member function:

```
lvColumn.iSubItem = 0;
lvColumn.pszText = "Column 0";
m_listView.InsertColumn(0, &amlvColumn);
```

Setting `iSubItem` to 0 indicates that the program is creating the first column. (Column numbers are zero-based like an array's indexes.) Finally, the program sets `pszText` to the column's label and calls `InsertColumn()` to add the column to the list view control. `InsertColumn()`'s two arguments are the column's index and a pointer to the `LV_COLUMN` structure.

The two subitem columns are created similarly, as shown in Listing 10.11.

---

**Listing 10.11 LST10\_11.CPP—Creating the Subitem Columns**

---

```
lvColumn.iSubItem = 1;
lvColumn.pszText = "Column 1";
m_listView.InsertColumn(1, &amlvColumn);
lvColumn.iSubItem = 2;
lvColumn.pszText = "Column 2";
m_listView.InsertColumn(1, &amlvColumn);
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Creating the List View's Items

With the columns created, it's time to create the items that will be listed in the columns when the control is in its report view. Creating items is not unlike creating columns. As with columns, Visual C++ defines a structure that you must initialize and pass to the function that creates the items. This structure is called `LV_ITEM` and is defined, as shown in Listing 10.12.

### Listing 10.12 LST10\_12.CPP—The `LV_ITEM` Structure

```
typedef struct _LV_ITEM
{
    UINT    mask;           // Flags indicating valid fields
    int     iItem;          // Item index
    int     iSubItem;       // Sub-item index
    UINT    state;          // Item's current state
    UINT    stateMask;      // Valid item states.
    LPSTR   pszText;        // Address of string buffer
    int     cchTextMax;     // Size of string buffer
    int     iImage;         // Image index for this item
    LPARAM  lParam;        // Additional information as a 32-bit value
} LV_ITEM;
```

In the `LV_ITEM` structure, the `mask` member specifies which other members of the structure are valid. The flags you can use are `LVIF_IMAGE` (`iImage` is valid), `LVIF_PARAM` (`lParam` is valid), `LVIF_STATE` (`state` is valid), and `LVIF_TEXT` (meaning `pszText` is valid).

The `iItem` member is the index of the item, which you can think of as the row number in report view (although the position of the items can change when they're sorted). Each item has a unique index. The `iSubItem` member is the index of the subitem if this structure is defining a subitem. You can think of this value as the number of the column in which the item will appear. If you're defining the main item (the first column), this value should be 0.

The `state` and `stateMask` members hold the item's current state and the item's valid states, which can be one or more of `LVIS_CUT` (the item is selected for cut and paste), `LVIS_DROPHILITED` (the item is a highlighted drop target), `LVIS_FOCUSED` (the item has the focus), and `LVIS_SELECTED` (the item is selected).

The `pszText` member is the address of a string buffer. When using the `LV_ITEM` structure to create an item, the string buffer contains the item's text. When obtaining information about the item, `pszText` is the buffer where the information will be stored, and `cchTextMax` is the size of the buffer. If `pszText` is set to `LPSTR_TEXTCALLBACK`, the item uses the callback mechanism. Finally, the `iImage` member is the index of the item's icon in the small icon and large icon image lists. If set to `I_IMAGECALLBACK`, the `iImage` member indicates that the item uses the callback mechanism.

In Win95 Controls App's `CreateListView()` function, the program starts initializing the `LV_ITEM` structure, as shown in Listing 10.13.

### Listing 10.13 LST10\_13.CPP—Initializing the LV\_ITEM Structure

---

```
LV_ITEM lvItem;  
lvItem.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_STATE;  
lvItem.state = 0;  
lvItem.stateMask = 0;  
lvItem.iImage = 0;
```

---

The values being set in Listing 10.13 will be the same for every item created, so they are done first and then are not changed as the items are created.

Now, the program can start creating the items that will be displayed in the list view control. Listing 10.14 shows how the program creates the first item.

### Listing 10.14 LST10\_14.CPP—Creating a List View Item

---

```
lvItem.iItem = 0;  
lvItem.iSubItem = 0;  
lvItem.pszText = "Item 0";  
m_listView.InsertItem(&amlvItem);
```

---

In Listing 10.14, the program sets the item and subitem indexes to 0 and sets the item's text to "Item 0." The program then calls the CListCtrl class's InsertItem() member function to add the item to the list view control. This function's single argument is the address of the LV\_ITEM structure that contains the information about the item to be created.

At this point, the list view control has three columns and one item created. However, this single item's subitems display nothing unless you initialize them. Win95 Controls App initializes the first set of subitems like this:

```
m_listView.SetItemText(0, 1, "Sub Item 0.1");  
m_listView.SetItemText(0, 2, "Sub Item 0.2");
```

The SetItemText() function takes three arguments, which are the item index, the subitem index, and the text to which to set the item or subitem.

As Listing 10.15 shows, the program then creates two more items along with the items' associated subitems.

### Listing 10.15 LST10\_15.CPP—Creating Additional Items and Subitems

---

```
lvItem.iItem = 1;  
lvItem.iSubItem = 0;  
lvItem.pszText = "Item 1";  
m_listView.InsertItem(&amlvItem);  
m_listView.SetItemText(1, 1, "Sub Item 1.1");  
m_listView.SetItemText(1, 2, "Sub Item 1.2");  
lvItem.iItem = 2;  
lvItem.iSubItem = 0;  
lvItem.pszText = "Item 2";  
m_listView.InsertItem(&amlvItem);  
m_listView.SetItemText(2, 1, "Sub Item 2.1");  
m_listView.SetItemText(2, 2, "Sub Item 2.2");
```

---

---

**Note:**

Make sure to insert a new item before setting the item text. Items that have not been inserted into the list view control will not accept text.

---

## Manipulating the List View

You can set a list view control to four different types of views: small icon, large icon, list, and report. In Explorer, for example, the toolbar features buttons that you can click to change the view, or you can select the view from the View menu. Although Win95 Controls App doesn't have a snazzy toolbar like Explorer, it does include four buttons that you can click to change the view. Those buttons are created in the `CreateListView()` function as shown in Listing 10.16.

### Listing 10.16 LST10\_16.CPP—Creating the View Buttons

---

```
m_smallButton.Create("Small", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 120, 450, 140), this, 106);
m_largeButton.Create("Large", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 145, 450, 165), this, 107);
m_listButton.Create("List", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 170, 450, 190), this, 108);
m_reportButton.Create("Report", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 195, 450, 215), this, 109);
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

These buttons are associated with entries in the view window’s message map with the message response functions OnSmall(), OnLarge(), OnList(), and OnReport(). In other words, when the user clicks one of these buttons, its matching function gets called, and the program changes the list view control to the requested view type. For example, when the user clicks the Small button, the function shown in Listing 10.17 changes the view to the list view.

**Listing 10.17 LST10\_17.CPP—Changing to the List View**

```
void CWin95View::OnSmall()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
        WS_VISIBLE | WS_CHILD | WS_BORDER |
        LVS_SMALLICON | LVS_EDITLABELS);
}
```

The SetWindowLong() function sets a window’s attribute. Its arguments are the window’s handle, a flag that specifies the value to be changed, and the new value. In this case, the GWL\_STYLE flag specifies that the window’s style should be changed to the style given in the third argument. Changing the list view control’s style (in the preceding code, the new view style is LVS\_SMALLICON) changes the type of view it displays.

Besides changing the view, there are a number of other features you can program for your list view controls. When the user does something with the control, Windows sends a WM\_NOTIFY message to the parent window. By responding to these notifications, you can give your list view control its various capabilities.

The most common notifications sent by a list view control are:

Notification	Purpose
LVN_COLUMNCLICK	Indicates that the user clicked a column header
LVN_BEGINLABELEDIT	Indicates that the user is about to edit an item’s label
LVN_ENDLABELEDIT	Indicates that the user is ending the label editing process

If you haven't discovered it yet, you can edit the labels of the items in Win95 Controls App's list view items. This works by the program's capturing and handling the notification messages sent by the list view control. To capture the notification messages, just override the window's OnNotify() function.

The three parameters received by OnNotify() are the message's WPARAM and LPARAM values and a pointer to a result code. In the case of a WM\_NOTIFY message coming from a list view control, the WPARAM is the list view control's ID. And, if the WM\_NOTIFY message is the LVN\_BEGINLABELEDIT or LVN\_ENDLABELEDIT notifications, the LPARAM is a pointer to a LV\_DISPINFO structure, which itself contains NMHDR and LV\_ITEM structures. You use the information in these structures to manipulate the item that the user is trying to edit.

In OnNotify(), the program first casts the lParam parameter to an LV\_DISPINFO structure, like this:

```
LV_DISPINFO* lv_dispInfo = (LV_DISPINFO*) lParam;
```

Next, the program checks whether the function is receiving a LVN\_BEGINLABELEDIT notification, like this:

```
if (lv_dispInfo->hdr.code == LVN_BEGINLABELEDIT)
```

If the notification is LVN\_BEGINLABELEDIT, your program can do whatever preediting initialization it needs to do. In Win95 Controls App, the function shows you how to get a pointer to the edit control being used to edit the label:

```
CEdit* pEdit = m_listView.GetEditControl();
```

The program, however, doesn't actually do anything with the control.

When handling label editing, the other notification to watch out for is LVN\_ENDLABELEDIT, which this particular application does like this:

```
else if (lv_dispInfo->hdr.code == LVN_ENDLABELEDIT)
```

When the program receives the LVN\_ENDLABELEDIT notification, the user has finished editing the label, either by typing the new label or by canceling the editing process. If the user has canceled the process, the LV\_DISPINFO structure's item.pszText member will be NULL, or the item.iItem member will be -1. In this case, you need do nothing more than ignore the notification. If, however, the user completed the editing process, the program must copy the new label to the item's text, which OnNotify() does like this:

```
m_listView.SetItemText(lv_dispInfo->item.iItem,  
    0, lv_dispInfo->item.pszText);
```

The CListCtrl object's SetItemText() member function requires three arguments: the item index, the subitem index, and the new text. As you can see, all of the information you need is stored in the LV\_DISPINFO structure.

---

**Note:**

There are a lot of other things you can do with a list view control. You can learn more about these powerful controls in your Visual C++ online documentation. The ROWLIST sample program in the DevStudio\VC\Samples\Mfc\General\ directory, for example, might be a good place to start.

---

## The Tree View Control

In the preceding section, you learned how to use the list view control to organize the display of many items in a window. The list view control enables you to display items both as objects in a window and objects in a report organized into columns. Often, however, the data you'd like to organize for your application's user is best placed into a hierarchical view, where elements of the data are shown as they relate to each other. A good example of such a hierarchical display is the directory tree used by Windows to display directories on the hard disk and the files that they contain.

As is the case with other useful controls, Windows 95 includes the tree view control as one of its common controls. MFC provides access to this control through its CTreeCtrl class. This versatile control enables you to display data in various ways, all the while retaining the hierarchical relationship between the data objects in the view.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

+ [Advanced](#)[Search](#)+ [Search Tips](#)BROWSE  
BY TOPIC

[an error occurred while processing this directive]

If you'd like to see an example of a tree view control, just open the Windows 95 Explorer (see Figure 10.11). The left side of the window shows how the tree view control organizes objects in a window. (The right side of the window contains a listview control, which you learned about in the previous section). In the figure, the tree view displays not only the storage devices on the computer, but also the directories on those devices. The tree clearly shows the hierarchical relationship between the devices, directories, and files and enables the user to open and close branches on the tree to explore it at a different level. To add a tree view control to Win95 Controls App, complete the steps that follow Figure 10.11.



The complete source code and executable file for this final part of the Win95 application can be found in the CHAP10\Win95 directory of this book's CD-ROM.

1. Load the WIN95VIEW.CPP file, and add the following line to the OnDraw() member function, right after the line `pDC->TextOut(160, 102, "List-view control")`, which you placed there previously.

```
pDC->TextOut(20, 240, "Tree View Control");
```

2. Add the lines shown in Listing 10.18 to the end of the CreateTreeView() function.



**FIG. 10.11** A tree view control shows a hierarchical relationship between items.

#### Listing 10.18 LST10\_18.CPP—New Lines for the CreateTreeView() Function

```
// Create the Tree View control.
m_treeView.Create(WS_VISIBLE | WS_CHILD | WS_BORDER |
    TVS_HASLINES | TVS_LINESATROOT | TVS_HASBUTTONS |
    TVS_EDITLABELS, CRect(20, 260, 160, 360), this, 110);
m_treeView.SetImageList(&m_treeImageList, TVSIL_NORMAL);

// Create the root item.
TV_ITEM tvItem;
tvItem.mask =
    TVIF_TEXT | TVIF_IMAGE | TVIF_SELECTEDIMAGE;
tvItem.pszText = "Root";
tvItem.cchTextMax = 4;
tvItem.iImage = 0;
tvItem.iSelectedImage = 0;
TV_INSERTSTRUCT tvInsert;
```

```

tvInsert.hParent = TVI_ROOT;
tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
HTREEITEM hRoot = m_treeView.InsertItem(&amptvInsert);

// Create the first child item.
tvItem.pszText = "Child Item 1";
tvItem.cchTextMax = 12;
tvItem.iImage = 1;
tvItem.iSelectedImage = 1;
tvInsert.hParent = hRoot;
tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
HTREEITEM hChildItem = m_treeView.InsertItem(&amptvInsert);

// Create a child of the first child item.
tvItem.pszText = "Child Item 2";
tvItem.cchTextMax = 12;
tvItem.iImage = 2;
tvItem.iSelectedImage = 2;
tvInsert.hParent = hChildItem;
tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
m_treeView.InsertItem(&amptvInsert);

// Create another child of the root item.
tvItem.pszText = "Child Item 3";
tvItem.cchTextMax = 12;
tvItem.iImage = 1;
tvItem.iSelectedImage = 1;
tvInsert.hParent = hRoot;
tvInsert.hInsertAfter = TVI_LAST;
tvInsert.item = tvItem;
m_treeView.InsertItem(&amptvInsert);

```

---

**3.** Add the lines shown in Listing 10.19 near the end of the OnNotify() function, right before the line return CView::OnNotify(wParam, lParam, pResult).

---

**Listing 10.19 LST10\_19.CPP—New Lines for the OnNotify() Function**

---

```

TV_DISPINFO* tv_dispInfo = (TV_DISPINFO*) lParam;

if (tv_dispInfo->hdr.code == TVN_BEGINLABELEDIT)
{
    CEdit* pEdit = m_treeView.GetEditControl();
    // Manipulate edit control here.
}
else if (tv_dispInfo->hdr.code == TVN_ENDLABELEDIT)
{
    if (tv_dispInfo->item.pszText != NULL)
    {

```

```

        m_treeView.SetItemText(tv_dispInfo->item.hItem,
                               tv_dispInfo->item.pszText);
    }
}

```

4. Load the view class's header file (WIN95VIEW.H), and add the line that follows to the class's Attributes section, right after the line `CButton m_reportButton`, which you placed there previously.

```
CTreeCtrl m_treeView;
```

You have now completed the final part of the Win95 Controls App. Compile and link the application by selecting the Build button in the toolbar, by selecting Developer Studio's Build, Build command, or by pressing F7. When you run the application now, you see the window shown in Figure 10.12.



**FIG. 10.12** Here's the Win95 Controls App with its new tree view control.

The Win95 Controls App application now contains a tree view control. You can click the tree's various nodes to expose new levels of the tree (see Figure 10.13). You can even edit the labels of the items in the tree. To do this, select an item and then click it. An edit box appears into which you can type the new label.



**FIG. 10.13** By clicking nodes in the tree, you can expose the tree's child nodes.

## Creating the Tree View

In the Win95 Controls App application, the tree view control is created in the `CreateTreeView()` local member function, which the program calls from the view class's `OnCreate()` function. In `CreateTreeView()`, the program first creates the image list that holds the icons used with each item in the view. Listing 10.20 shows how the program creates the image list.

### Listing 10.20 LST10\_20.CPP—Creating the Tree View Control's Image List

```

m_treeImageList.Create(13, 13, FALSE, 3, 0);
HICON hIcon = ::LoadIcon(AfxGetResourceHandle(),
                        MAKEINTRESOURCE(IDI_ICON3));
m_treeImageList.Add(hIcon);
hIcon = ::LoadIcon(AfxGetResourceHandle(),
                  MAKEINTRESOURCE(IDI_ICON4));
m_treeImageList.Add(hIcon);
hIcon = ::LoadIcon(AfxGetResourceHandle(),
                  MAKEINTRESOURCE(IDI_ICON5));

```

```
m_treeImageList.Add(hIcon);
```

---

Now that the program has created the image list, it can create the tree view control, by calling the CTreeCtrl class's Create() member function:

```
m_treeView.Create(WS_VISIBLE | WS_CHILD | WS_BORDER |  
    TVS_HASLINES | TVS_LINESATROOT | TVS_HASBUTTONS |  
    TVS_EDITLABELS, CRect(20, 260, 160, 360), this, 110);
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 10.3 Tree View Control Styles

Style	Description
TVS_DISABLEDRAHDROP	Disables drag-and-drop operations
TVS_EDITLABELS	Enables user to edit labels
TVS_HASBUTTONS	Gives each parent item a button
TVS_HASLINES	Adds lines between items in the tree
TVS_LINESATROOT	Adds a line between the root and child items
TVS_SHOWSELALWAYS	Forces a selected item to stay selected when losing focus

Initializing the Tree View

Like the list view control, a tree view control requires some hefty setup work. As you already know, the tree view control can use an image list for item icons. A tree view control also must contain item objects that your program creates and adds to the control. To initialize a tree view control, you must complete the following steps.

1. Create the tree view control.
2. Associate the control with its image list (optional).
3. Create the root and child items that will be displayed in the control.

Win95 Controls App associates the tree view control with its image list like this:

```
m_treeView.SetImageList(&amp;treeImageList, TVSIL_NORMAL);
```

The creation of the tree view controls root and child items is covered in the following section.

Creating the Tree View’s Items

Creating items for a tree view control is much like doing the same thing for a list view control. As with the list view, Visual C++ defines a structure that you must initialize and pass to the function that creates the items. This structure is called TV\_ITEM and is defined as shown in Listing 10.21.

Listing 10.21 LST10\_21.CPP—The TV\_ITEM Structure

```

typedef struct _TV_ITEM
{
    UINT          mask;
    HTREEITEM     hItem;
    UINT          state;
    UINT          stateMask;
    LPSTR         pszText;
    int           cchTextMax;

```

```

        int          iImage;
int      iSelectedImage;
        int          cChildren;
        LPARAM       lParam;
    } TV_ITEM;

```

---

In the TV\_ITEM structure, the mask member specifies which other members of the structure are valid. The flags you can use are:

Flags	Other Members
TVIF_CHILDREN	cChildren is valid
TVIF_HANDLE	hItem is valid
TVIF_IMAGE	iImage is valid
TVIF_PARAM	lParam is valid
TVIF_SELECTEDIMAGE	iSelectedImage is valid
TVIF_STATE	state and stateMask are valid
TVIF_TEXT	pszText and cchTextMax are valid

The hItem member is the handle of the item, whereas the state and stateMask members hold the item's current state and the item's valid states, which can be one or more of

- TVIS\_BOLD
- TVIS\_CUT
- TVIS\_DROPHILITED
- TVIS\_EXPANDED
- TVIS\_EXPANDEDONCE
- TVIS\_FOCUSED
- TVIS\_OVERLAYMASK
- TVIS\_SELECTED
- TVIS\_STATEIMAGEMASK
- TVIS\_USERMASK

Please check your Visual C++ online documentation for the meanings of these flags.

The pszText member is the address of a string buffer. When you're using the LV\_ITEM structure to create an item, the string buffer contains the item's text. When you're obtaining information about the item, pszText is the buffer where the information will be stored, and cchTextMax is the size of the buffer. If pszText is set to LPSTR\_TEXTCALLBACK, the item uses the callback mechanism. Finally, the iImage member is the index of the item's icon in the image list. If set to I\_IMAGECALLBACK, the iImage member indicates that the item uses the callback mechanism.

The iSelectedImage member is the index of the icon in the image list that represents the item when the item is selected. As with iImage, if this member is set to I\_IMAGECALLBACK, the iSelectedImage member indicates that the item uses the callback mechanism. Finally, cChildren specifies whether or not there are child items associated with the item.

Besides the TV\_ITEM structure, you must initialize a TV\_INSERTSTRUCT structure that holds information about how to insert the new structure into the tree view control. That structure is declared,

as shown in Listing 10.22.

### Listing 10.22 LST10\_22.CPP—The TV\_INSERTSTRUCT Structure

---

```
typedef struct _TV_INSERTSTRUCT
{
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
    TV_ITEM    item;
} TV_INSERTSTRUCT;
```

---

[Previous](#) [Table of Contents](#) [Next](#)



Brief    Full  
• [Advanced Search](#)  
• [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Here, Create()'s four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. The CTreeCtrl class, of which m\_treeView is an object, defines special styles to be used with list view controls. Table 10.3 lists these special styles.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

In this structure, hParent is the handle to the parent tree view item. A value of NULL or TVI\_ROOT specifies that the item should be placed at the root of the tree. The hInsertAfter member specifies the handle of the item after which this new item should be inserted. It can also be one of the flags TVI\_FIRST (beginning of the list), TVI\_LAST (end of the list), or TVI\_SORT (alphabetical order). Finally, the item member is the TV\_ITEM structure containing information about the item to be inserted into the tree.

In Win95 Controls App's CreateTreeView() function, the program initializes the TV\_ITEM structure for the root item (the first item in the tree) as shown in Listing 10.23.

### Listing 10.23 LST10\_23.CPP—Creating the Root Item

```
TV_ITEM tvItem;
tvItem.mask =
    TVIF_TEXT | TVIF_IMAGE | TVIF_SELECTEDIMAGE;
tvItem.pszText = "Root";
tvItem.cchTextMax = 4;
tvItem.iImage = 0;
tvItem.iSelectedImage = 0;
TV_INSERTSTRUCT tvInsert;
tvInsert.hParent = TVI_ROOT;
tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
HTREEITEM hRoot = m_treeView.InsertItem(&tvInsert);
```

As you can see, the CTreeCtrl member function InsertItem() actually inserts the item into the tree view control. Its single argument is the address of the TV\_INSERTSTRUCT structure.

The program inserts the remaining items into the tree view control as shown in Listing 10.24.

### Listing 10.24 LST10\_24.CPP—Inserting Child Items into the Tree View Control

```
// Create the first child item.
tvItem.pszText = "Child Item 1";
tvItem.cchTextMax = 12;
tvItem.iImage = 1;
tvItem.iSelectedImage = 1;
tvInsert.hParent = hRoot;
```



```

tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
HTREEITEM hChildItem = m_treeView.InsertItem(&amptvInsert);
// Create a child of the first child item.
tvItem.pszText = "Child Item 2";
tvItem.cchTextMax = 12;
tvItem.iImage = 2;
tvItem.iSelectedImage = 2;
tvInsert.hParent = hChildItem;
tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
m_treeView.InsertItem(&amptvInsert);
// Create another child of the root item.
tvItem.pszText = "Child Item 3";
tvItem.cchTextMax = 12;
tvItem.iImage = 1;
tvItem.iSelectedImage = 1;
tvInsert.hParent = hRoot;
tvInsert.hInsertAfter = TVI_LAST;
tvInsert.item = tvItem;
m_treeView.InsertItem(&amptvInsert);

```

---

## Manipulating the Tree View

Just as with the list view control, you can edit the labels of the items in Win95 Controls App's tree view items. Also like the list view control, this process works by the program's capturing and handling (in `OnNotify()`) the notification messages sent by the tree view control. In the case of a `WM_NOTIFY` message coming from a tree view control, the `WPARAM` is the list view control's ID. And, if the `WM_NOTIFY` message is the `TVN_BEGINLABELEDIT` or `TVN_ENDLABELEDIT` notifications, the `LPARAM` is a pointer to a `TV_DISPINFO` structure, which itself contains `NMHDR` and `TV_ITEM` structures. You use the information in these structures to manipulate the item the user is trying to edit. As you can see in Listing 10.25, `OnNotify()` handles the tree view notifications almost exactly the same way as the list view notifications. The only difference is the names of the structures used.

### Listing 10.25 LST10\_25.CPP—Handling Tree View Notifications

---

```

TV_DISPINFO* tv_dispInfo = (TV_DISPINFO*) lParam;
if (tv_dispInfo->hdr.code == TVN_BEGINLABELEDIT)
{
    CEdit* pEdit = m_treeView.GetEditControl();
    // Manipulate edit control here.
}
else if (tv_dispInfo->hdr.code == TVN_ENDLABELEDIT)
{
    if (tv_dispInfo->item.pszText != NULL)
    {
        m_treeView.SetItemText(tv_dispInfo->item.hItem,

```

```
        tv_dispInfo->item.pszText);  
    }  
}
```

---

The tree view control sends a number of different notification messages, including

- TVN\_BEGINDRAG
- TVN\_BEGINLABELEDIT
- TVN\_BEGINRDRAG
- TVN\_DELETEITEM
- TVN\_ENDLABELEDIT
- TVN\_GETDISPINFO
- TVN\_ITEMEXPANDED
- TVN\_ITEMEXPANDING
- TVN\_KEYDOWN
- TVN\_SELCHANGED
- TVN\_SELCHANGING
- TVN\_SETDISPINFO

Now that you know about image list, list, and tree view controls, you're ready to move on to toolbars and status bars, which just happen to be the subject of the next chapter.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

## Chapter11

# Toolbars and Status Bars

- How to create a basic toolbar
- How to create a dockable tool-bar
- How to create a custom toolbar
- How to create a basic status bar
- How to create a custom status bar

Building a good user interface is half the battle of programming a Windows application. Luckily, MFC supplies an amazing amount of help in creating an application that supports all of the expected user-interface elements, including menus, dialog boxes, toolbars, and status bars. The subjects of menus and dialog boxes are covered earlier in this book, in Chapter 6, “Using Menus,” and in Chapter 7, “Programming Dialog Boxes.” In this chapter, you learn how to get the most out of toolbars and status bars.

## Working with Toolbars

Although you can add a toolbar to your application with AppWizard, you still need to use a little programming polish to get things just right. This is because every application is different and AppWizard can create only the most generally useful toolbar for most applications. When you create your own toolbars, however, you’ll almost certainly want to add or delete buttons to support your application’s unique command set.

For example, when you create a standard AppWizard application with a toolbar, AppWizard creates the toolbar shown in Figure 11.1. This toolbar provides buttons for the commonly used commands in the File and Edit menus, as well as a button for displaying the About dialog box. But what if your application won’t support these commands or adds additional commands? It’s up to you to modify the default toolbar to fit your application.

The truth is that modifying an AppWizard-generated application’s toolbar is an easy task, especially if you know how toolbars work. Because adding a toolbar to any application is also fairly easy, in this chapter you’ll put together an application that creates and manipulates its toolbar without the help of AppWizard. After you’ve finished this chapter, you’ll have no problem dealing with toolbars in AppWizard or non-AppWizard applications.



**FIG. 11.1** The default toolbar provides buttons for commonly used commands.

## Introducing the Toolbar Application



Before you start digging into the details of toolbar creation and manipulation, take a look at the sample program, an application called `Toolbar` that you'll find in the `Chap11\Toolbar1` folder on this book's CD-ROM. When you run the application, you see the window shown in Figure 11.2. The toolbar sports three buttons. Just click a button to change the color of the rectangle displayed in the application's window.



**FIG. 11.2** The `Toolbar` application has a three-button toolbar.

Another cool thing about the `Toolbar` application is the way that you can drag the toolbar from its resting place below the menu bar and drop it elsewhere on the screen, changing it into a toolbox (see Figure 11.3). Go ahead and try it. After you have the toolbox on the screen, drag it down to the bottom of the application's window. You can dock it there, as shown in Figure 11.4, creating a toolbar at the bottom of the window. (You can, of course, also dock the toolbar back at the top of the window, leaving it where it started.)



**FIG. 11.3** The `Toolbar` application features a docking toolbar that can be changed into a toolbox.



**FIG. 11.4** You can even dock the toolbar at the bottom of the window.

Finally, you can hide or display the toolbar by choosing the application's `View, Toolbar` command. The source code for the `Toolbar` application is shown in Listings 11.1 through 11.4.

### Listing 11.1 `toolbar.h`—The `CToolbarApp` Class's Header File

---

```

////////////////////////////////////
// TOOLBAR.H: Header file for the CToolBarApp class, which
//             represents the application object.
////////////////////////////////////

class CToolBarApp : public CWinApp
{
public:
    CToolBarApp();

    // Virtual function overrides.
    BOOL InitInstance();
};

```

---

### **Listing 11.2 toolbar.cpp—The CToolBarApp Class’s Implementation File**

---

```

////////////////////////////////////
// TOOLBAR.CPP: Implementation file for the CToolBarApp,
//             class, which represents the application
//             object.
////////////////////////////////////

#include <afxwin.h>
#include "toolbar.h"
#include "mainfrm.h"

// Global application object.
CToolbarApp ToolbarApp;

////////////////////////////////////
// Construction/Destruction.
////////////////////////////////////
CToolbarApp::CToolbarApp()
{
}

////////////////////////////////////
// Overrides
////////////////////////////////////
BOOL CToolBarApp::InitInstance()
{
    m_pMainWnd = new CMainFrame();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

```

---

### Listing 11.3 mainfrm.h—The CMainFrame Class's Header File

---

```
////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//           represents the application's main window.
////////////////////////////////////

#include <afxext.h>

enum {Red, Green, Blue};

class CMainFrame : public CFrameWnd
{
// Protected data members.
protected:
    CToolBar m_toolbar;
    int m_color;

// Constructor and destructor.
public:
    CMainFrame();
    ~CMainFrame();

// Overrides.
protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Message map functions.
public:
    afx_msg void OnPaint();
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnRed();
    afx_msg void OnGreen();
    afx_msg void OnBlue();
    afx_msg void OnViewToolBar();
    afx_msg void OnUpdateViewToolBarUI(CCmdUI* pCmdUI);

    DECLARE_MESSAGE_MAP()
};
```

---

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

## Listing 11.4 mainfrm.cpp—The CMainFrame Class's Implementation File

```

////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//               class, which represents the application's
//               main window.
////////////////////////////////////

#include <afxwin.h>
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE( )
    ON_WM_PAINT( )

    ON_COMMAND(ID_RED, OnRed)
    ON_COMMAND(ID_GREEN, OnGreen)
    ON_COMMAND(ID_BLUE, OnBlue)
    ON_COMMAND(ID_VIEW_TOOLBAR, OnViewToolbar)

    ON_UPDATE_COMMAND_UI(ID_VIEW_TOOLBAR, OnUpdateViewToolbarUI)
END_MESSAGE_MAP( )

////////////////////////////////////
// CMainFrame: Construction and destruction.
////////////////////////////////////
CMainFrame::CMainFrame( )
{
    // Create the main window. The WS_CLIPCHILDREN style
    // ensures that drawing in the window won't erase
    // parts of the toolbar.
    Create(NULL, "Toolbar App", WS_OVERLAPPEDWINDOW |
        WS_CLIPCHILDREN, rectDefault,
        NULL, MAKEINTRESOURCE(IDR_MENU1));

    // Set the initial rectangle color to red.
    m_color = Red;
}

CMainFrame::~CMainFrame( )
{

```



```

}

////////////////////////////////////
// Overrides.
////////////////////////////////////
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Set size of the main window.
    cs.cx = 365;
    cs.cy = 440;

    // Call the base class's version.
    BOOL returnCode = CFrameWnd::PreCreateWindow(cs);

    return returnCode;
}

////////////////////////////////////
// Message map functions.
////////////////////////////////////
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // Call the base class's OnCreate().
    CFrameWnd::OnCreate(lpCreateStruct);

    // Create and load the toolbar.
    m_toolbar.Create(this, WS_CHILD | WS_VISIBLE | CBRS_TOP);
    m_toolbar.LoadToolBar(IDR_TOOLBAR1);

    // Set the toolbar's styles.
    DWORD styles = m_toolbar.GetBarStyle();
    styles |= CBRS_TOOLTIPS | CBRS_FLYBY;
    m_toolbar.SetBarStyle(styles);

    // Enable toolbar docking.
    m_toolbar.EnableDocking(CBRS_ALIGN_TOP | CBRS_ALIGN_BOTTOM);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_toolbar);

    return 0;
}

void CMainFrame::OnPaint()
{
    CBrush* newBrush;
    CPaintDC paintDC(this);

    // Create the appropriate color brush.
    if (m_color == Red)
        newBrush = new CBrush(RED);
    else if (m_color == Green)
        newBrush = new CBrush(GREEN);
    else

```

```

        newBrush = new CBrush(RGB(0,0,255));

        // Select the new brush into the DC.
        CBrush* oldBrush = paintDC.SelectObject(newBrush);

        // Draw the rectangle.
        paintDC.Rectangle(50, 70, 300, 340);

        // Restore the DC and delete the new brush.
        paintDC.SelectObject(oldBrush);
        delete newBrush;
    }

void CMainFrame::OnRed()
{
    // Set the selected color to Red
    // and redraw the window.
    m_color = Red;
    Invalidate();
}

void CMainFrame::OnGreen()
{
    // Set the selected color to Green
    // and redraw the window.
    m_color = Green;
    Invalidate();
}

void CMainFrame::OnBlue()
{
    // Set the selected color to Blue
    // and redraw the window.
    m_color = Blue;
    Invalidate();
}

void CMainFrame::OnViewToolbar()
{
    // Determine whether the toolbar is visible.
    BOOL visible = m_toolbar.GetStyle() & WS_VISIBLE;

    // Show or hide the toolbar.
    ShowControlBar(&m_toolbar, !visible, FALSE);
}

void CMainFrame::OnUpdateViewToolbarUI(CCmdUI* pCmdUI)
{
    // Determine whether the toolbar is visible.
    BOOL visible = m_toolbar.GetStyle() & WS_VISIBLE;

    // Check or uncheck the Toolbar menu item.
    pCmdUI->SetCheck(visible);
}

```

}

## Creating the Toolbar Resource

The first step in adding a toolbar to an application is to create the toolbar's resource. You can do this easily by using Developer Studio's toolbar editor. To create a toolbar resource, load or start a project and complete the following steps.

1. Choose the Insert, Resource command from Developer Studio's menu bar. The Insert Resource dialog box appears, as shown in Figure 11.5.



**FIG. 11.5** You can choose a type of resource from the Insert resource dialog box.

2. In the Resource Type box, choose Toolbar, and then click New. Developer Studio creates a blank toolbar and loads it into the toolbar editor (see Figure 11.6).



**FIG. 11.6** Developer Studio's toolbar editor enables you to customize your application's toolbar.

3. Use the toolbar editor's tools to draw an icon on the blank toolbar button. When you do, a new toolbar button appears next to the button on which you drew an icon, as shown in Figure 11.7.
4. After drawing the button's icon, double-click the button's image in the toolbar. The Toolbar Button Properties sheet appears.
5. Type the button's resource ID in the ID box, and type the button's prompt text in the Prompt box, as shown in Figure 11.8.



**FIG. 11.7** A new button appears when you draw on the original blank button.

---

### Tip:

The prompt is comprised of two strings separated by a newline character (`\n`). The first string is the hint text that appears in the application's status bar (if the application has a status bar), and the second string is the text for the button's tool tip, which appears whenever the user holds the mouse pointer over the button for a second or two.

---



**FIG. 11.8** The Toolbar Button Properties sheet enables you to set a button's ID and prompt.

6. Repeat steps 3 through 5 for each button that you want to add to the toolbar.

## Creating Toolbar Message Response Functions

Now that you have created your toolbar buttons, you have to tell the application what to do when the user chooses one of the buttons. You do this by associating, in the application's message map, a message response function with each button's resource ID.

- See "Responding to Windows Messages," for more information on message maps, **p. 68**

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Adding a message response function for a toolbar button is almost exactly like adding one for a menu item. First, you place an ON\_COMMAND macro in the message map for each toolbar button. Such an entry looks like this:

```
ON_COMMAND( ID_RED, OnRed )
```

The two arguments for the ON\_COMMAND message map macro are the command's ID and the name of the function that responds to a message with the given ID. In the example here, whenever the user clicks the toolbar button with the ID\_RED ID, the OnRed() message response function is called.

In the message response function, you write the program lines that will perform the actions necessary to respond to the command. In the case of the ID\_RED button, this means drawing a red rectangle in the window, as shown in Listing 11.5.

#### Listing 11.5 lst11\_05.cpp—Responding to a Toolbar Button

```
void CMainFrame::OnRed()
{
    // Set the selected color to Red
    // and redraw the window.
    m_color = Red;
    Invalidate();
}
```

In Listing 11.5, the program sets the member variable m\_color to indicate the selected color. Then a call to Invalidate() causes the window to be redrawn. In the program's OnPaint() function, the program uses m\_color to set the appropriate color before drawing the rectangle, as shown in Listing 11.6.

- See “Painting in an MFC Program,” for more information on drawing in a window, **p. 81**

#### Listing 11.6 lst11\_06.cpp—Drawing the Rectangle with the Selected Color

```
void CMainFrame::OnPaint()
{
    CBrush* newBrush;
    CPaintDC paintDC(this);

    // Create the appropriate color brush.
    if (m_color == Red)
        newBrush = new CBrush(RGB(255,0,0));
    else if (m_color == Green)
        newBrush = new CBrush(RGB(0,255,0));
```

```

else
    newBrush = new CBrush(RGB(0,0,255));

// Select the new brush into the DC.
CBrush* oldBrush = paintDC.SelectObject(newBrush);

// Draw the rectangle.
paintDC.Rectangle(50, 70, 300, 340);

// Restore the DC and delete the new brush.
paintDC.SelectObject(oldBrush);
delete newBrush;
}

```

---

#### Note:

If you haven't defined a message response function for a toolbar button, MFC disables the button when you run the application. This is also true for menu commands that have not yet been associated with a message response function. In fact, for all intents and purposes, toolbar buttons are menu commands.

---



---

#### Note:

Ordinarily, toolbar buttons duplicate menu commands, providing a quicker way for the user to select commonly used commands in the menus. In this case, the menu item and the toolbar button both represent the exact same command and you give both the same ID. Then, the same message response function is called whether or not the user selects the command from the menu bar or from the toolbar.

---

## Creating and Displaying a Toolbar

At this point, you know how to create a toolbar resource, as well as how to respond to the toolbar buttons in your program. You have only one tiny problem. You haven't yet displayed the toolbar in your application's window, which makes it mighty tough for the user to click one of its buttons!

You create your toolbar in the window class's `OnCreate()` function, which, after you add the `ON_WM_CREATE` macro to the window's message map, MFC calls in response to the Windows message `WM_CREATE`. The first step is to create an object of the MFC `CToolBar` class, like this:

```
m_toolbar.Create(this);
```

Here, `m_toolbar` is a data member of the `CMainFrame` window class. You call the toolbar object's `Create()` function with at least one argument, which is a pointer to the parent window. The `Create()` function, however, actually requires three arguments, two of which have default values. The function's signature looks like this:

```

BOOL Create( CWnd* pParentWnd, DWORD dwStyle =
    WS_CHILD | WS_VISIBLE | CBRS_TOP, UINT nID = AFX_IDW_TOOLBAR );

```

As you can see, the second argument is the toolbar's window styles, and the third argument is the toolbar's ID, which is important in applications that have multiple toolbars. (The application does, after all, need some way to tell one toolbar from another.) The default styles create a normal child-window toolbar that's positioned at the top of the parent window. You can, however, use other toolbar styles, which are described in Table 11.1. (The default toolbar ID is used internally by MFC and doesn't affect the ID you gave the toolbar when you created its resources.)

**Table 11.1 Toolbar Styles**

Style	Description
CBRS_BOTTOM	The toolbar is placed at bottom of the frame window.
CBRS_FLOATING	The toolbar is displayed in a floating window.
CBRS_FLYBY	The application's status bar can display the button's description.
CBRS_HIDE_INPLACE	The toolbar is hidden from the user.
CBRS_NOALIGN	The toolbar will not be repositioned when the parent window is resized.
CBRS_SIZE_DYNAMIC	The toolbar can be resized.
CBRS_SIZE_FIXED	The toolbar cannot be resized.
CBRS_TOOLTIPS	The toolbar can display tool tips.
CBRS_TOP	The toolbar is placed at top of the frame window.

[Previous](#)
[Table of Contents](#)
[Next](#)

[Products](#) | 
 [Contact Us](#) | 
 [About Us](#) | 
 [Privacy](#) | 
 [Ad Info](#) | 
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 + [Advanced Search](#)  
 + [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

After creating the CToolBar object, the program must load the toolbar from the application’s resources, like this:

```
m_toolbar.LoadToolBar( IDR_TOOLBAR1 );
```

The LoadToolBar() function’s single argument is the toolbar’s resource ID. In this example, the toolbar uses the default ID suggested by the Developer Studio when the toolbar resource was created.

**Note:**

Because the CToolBar class is declared in the AFXEXT.H file, you must add the line `#include <afxext.h>` to any file that accesses the CToolBar class. This is also true for status bars, which are represented by the MFC class CStatusBar.

**Setting the Toolbar’s Styles**

After you call the toolbar’s Create() and LoadToolBar() functions, the toolbar is displayed on the screen, fully functional. However, you might still want to tinker a bit with how the toolbar looks and acts. For example, the Toolbar sample program displays tool tips for the toolbar buttons. To enable tool tips, you have to change the toolbar’s default style. You can add the toolbar styles that you want when you call Create() to create the toolbar, like this:

```
m_toolbar.Create(this, WS_CHILD | WS_VISIBLE |
    CBRS_TOP | CBRS_TOOLTIPS);
```

Another way, however, is to use the toolbar’s member functions to change its styles after the toolbar has been created. This is the method used by the Toolbar example program. The first step is to get the styles that are already assigned to the toolbar. You do this by calling the toolbar object’s GetBarStyle() function:

```
DWORD styles = m_toolbar.GetBarStyle();
```

This function returns a double word containing the toolbar’s current styles. You can then add whatever styles you like by ORing the new styles with the styles returned by GetBarStyle(), like this:

```
styles |= CBRS_TOOLTIPS | CBRS_FLYBY;
```

This line adds the CBRS\_TOOLTIPS and CBRS\_FLYBY styles to the toolbar’s current styles that were returned in styles. The CBRS\_TOOLTIPS style enables the toolbar to display a



tool tip whenever the user places the mouse pointer over the toolbar button for a second or two, as shown in Figure 11.9. The `CBSR_FLYBY` style enables the application's status bar (of which the Toolbar application currently has none, but will later in this chapter, in the section "Working with Status Bars") to display a description of the button's command when the mouse pointer passes over the button.



**FIG. 11.9** Tool tips appear when the user places the mouse pointer over a toolbar button.

After adding the styles that you want, you hand the styles back to the toolbar by calling the toolbar object's `SetBarStyle()` function, like this:

```
m_toolbar.SetBarStyle(styles);
```

This function's single argument is the double word containing the toolbar's style flags.

## Enabling Toolbar Docking

One of the advanced features of an MFC toolbar is its capability to be docked and undocked from its window. When the user drags the toolbar from the top of the window and drops it elsewhere on the screen, the toolbar becomes a toolbox, which is a floating window containing the toolbar buttons. The user can also dock the toolbar back into the window wherever the application permits, usually at the top or bottom of the window. The user can choose the exact position of the toolbar's docking place: to the left, middle, right, or anywhere in between. Figure 11.10, for example, shows the Toolbar application's toolbar docked in the center of the upper portion of the window.



**FIG. 11.10** A dockable toolbar can be placed just about anywhere in the parent window.

To enable toolbar docking, you must first call the toolbar object's `EnableDocking()` function, like this:

```
m_toolbar.EnableDocking(CBSR_ALIGN_TOP | CBSR_ALIGN_BOTTOM);
```

The function's single argument is the styles that indicate where the toolbar can be docked. These styles can be any combination of `CBSR_ALIGN_TOP`, `CBSR_ALIGN_BOTTOM`, `CBSR_ALIGN_LEFT`, `CBSR_ALIGN_RIGHT`, and `CBSR_ALIGN_ANY`.

After you've enabled docking for the toolbar, you must do the same for the window in which the toolbar will be docked:

```
EnableDocking(CBSR_ALIGN_ANY);
```

The window object's `EnableDocking()` function requires similar style flags. In this case, however, you're determining not where any specific toolbar can be docked, but where *any* toolbar can be docked. This is a handy system whenever you want more than one toolbar. Each toolbar can be restricted to a given area, where the given area for all toolbars is determined by the window. Here's an example:

```
m_toolbar1.EnableDocking(CBRS_ALIGN_TOP);  
m_toolbar2.EnableDocking(CBRS_ALIGN_BOTTOM);  
EnableDocking(CBRS_ALIGN_TOP | CBRS_ALIGN_BOTTOM);
```

Here, toolbar 1 can be docked only at the top of the window, whereas toolbar 2 can be docked only at the bottom of the window. The window must enable docking on both the top and bottom of the window.

After enabling docking for both the toolbar and the window, the last step is to actually dock the toolbar by calling the window's `DockControlBar()` function, like this:

```
DockControlBar(&m_toolbar);
```

The `DockControlBar()` function's single argument is the address of the toolbar object to dock.

---

**Note:**

If you fail to call `DockControlBar()` for a dockable toolbar, the user will be unable to undock or dock the toolbar.

---

## Showing and Hiding the Toolbar

Like most applications with toolbars, in the Toolbar example application, the program enables the user to hide the toolbar in order to have more window space. The user can hide the toolbar by choosing the application's View, Toolbar command from the menu bar. To respond to this command, the program has a `OnViewToolbar()` message response function. The first thing that the program must do in this function is determine whether the toolbar is currently visible or hidden. A call to toolbar object's `IsVisible()` member function (inherited from the `CControlBar` class) provides this information:

```
BOOL visible = m_toolbar.IsVisible();
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The `IsVisible()` function returns `TRUE` if the toolbar is currently displayed and returns `FALSE` if the toolbar is hidden. All the application must do now is call the toolbar's `ShowControlBar()` function to hide or show the toolbar as appropriate:

```
ShowControlBar(&m_toolbar, !visible, FALSE);
```

This function's three arguments are

- The address of the toolbar object
- A Boolean value indicating whether the toolbar should be displayed (`TRUE`) or hidden (`FALSE`)
- A Boolean value indicating whether you want to delay showing the toolbar (`TRUE`) or show it immediately (`FALSE`)

As you can see in the previous line of code, the program displays or hides the toolbar by using the NOT operator on the Boolean value returned from `IsVisible()`. The NOT operator reverses the Boolean value so that if the toolbar is already displayed, the toolbar will be hidden, and vice versa.

You might have noticed that the `View, Toolbar` command of your application is checked when the toolbar is visible and unchecked when it's hidden. This is done by creating a command UI function for the command, as shown in Listing 11.7. (For more information on command UI functions, please refer back to Chapter 6, "Using Menus.")

### Listing 11.7 `lst11_07.cpp`—The Toolbar Command's Command UI Function

```
void CMainFrame::OnUpdateViewToolbarUI(CCmdUI* pCmdUI)
{
    // Determine whether the toolbar is visible.
    BOOL visible = m_toolbar.IsVisible();

    // Check or uncheck the Toolbar menu item.
    pCmdUI->SetCheck(visible);
}
```

The `OnUpdateViewToolbarUI()` function calls `IsVisible()` to determine the toolbar's visible state. The function then uses the returned Boolean value as the `SetCheck()` function's single argument. This places a checkmark next to the command when

the toolbar is visible and removes the checkmark when the toolbar is hidden.

## Creating Toolbar Radio Buttons

When you come right down to it, the buttons on the Toolbar application's toolbar should work like radio buttons. That is, because the buttons determine the state of the rectangle in the window—a mutually exclusive state in which the rectangle must always be one and only one of three colors—they should reflect that state by leaving the currently selected color's button depressed. Then, when the user selects a new color, the old color's button should pop up, and the new color's button should remain depressed.



The Toolbar2 application, which you'll find in the Chap11\Toolbar2 folder on this book's CD-ROM, adds this functionality to the program's toolbar. Figure 11.11 shows the application when it's first run. Because the rectangle begins red, the toolbar's red button is depressed. That button stays depressed until the user selects a new color.



**FIG. 11.11** The Toolbar2 application features radio-style buttons in its toolbar.

To create a radio-button toolbar, you use the class data member that tracks the currently selected button. As you know, the Toolbar2 application does this by creating an enumeration containing the possible button states, like this:

```
enum {Red, Green, Blue};
```

The application then declares a data member in which to store the currently selected color:

```
int m_color;
```

So that the program starts off with the correct button depressed, the window class's constructor sets this new member variable to its starting value:

```
m_color = Red;
```

In the toolbar buttons' message response functions, the program changes the value of `m_color` as appropriate.

The value of the `m_color` variable is used in a set of command UI functions created for each button, as shown in Listing 11.8:

### Listing 11.8 `lst11_08.cpp`—Setting the Radio-Button State

---

```
void CMainFrame::OnUpdateRedUI(CCmdUI* pCmdUI)
{
```

```

        // Press or unpress the toolbar button.
        pCmdUI->SetCheck(m_color == Red);
    }

void CMainFrame::OnUpdateGreenUI(CCmdUI* pCmdUI)
{
    // Press or unpress the toolbar button.
    pCmdUI->SetCheck(m_color == Green);
}

void CMainFrame::OnUpdateBlueUI(CCmdUI* pCmdUI)
{
    // Press or unpress the toolbar button.
    pCmdUI->SetCheck(m_color == Blue);
}

```

---

In this case, the `SetCheck()` function doesn't create a checkmark, but rather depresses the selected toolbar button and unpresses the other buttons.

To modify the `Toolbar1` application so that its toolbar uses radio buttons, perform the following steps:

The complete source code and executable file for the `Toolbar2` application can be found in the `Chap11\Toolbar2` directory on this book's CD-ROM.



1. Load `mainfrm.h`, and add the following lines to the message map function declarations, right after the line `afx_msg void OnUpdateViewToolbarUI(CCmdUI* pCmdUI):`

```
afx_msg void OnUpdateRedUI(CCmdUI* pCmdUI);
```

```
afx_msg void OnUpdateGreenUI(CCmdUI* pCmdUI);
```

```
afx_msg void OnUpdateBlueUI(CCmdUI* pCmdUI);
```

2. Load `mainfrm.cpp`, and add the following lines to the class's message map, right after the line `ON_UPDATE_COMMAND_UI(ID_VIEW_TOOLBAR, OnUpdateViewToolbarUI):`

```
ON_UPDATE_COMMAND_UI(ID_RED, OnUpdateRedUI)
```

```
ON_UPDATE_COMMAND_UI(ID_GREEN, OnUpdateGreenUI)
```

```
ON_UPDATE_COMMAND_UI(ID_BLUE, OnUpdateBlueUI)
```

3. Add the functions shown in Listing 11.9 to the end of the `mainfrm.cpp` file.

**Listing 11.9** `lst11_09.cpp`—New UI Functions for the Toolbar Application

---

```
void CMainFrame::OnUpdateRedUI(CCmdUI* pCmdUI)
{
    // Press or unpress the toolbar button.
    pCmdUI->SetCheck(m_color == Red);
}

void CMainFrame::OnUpdateGreenUI(CCmdUI* pCmdUI)
{
    // Press or unpress the toolbar button.
    pCmdUI->SetCheck(m_color == Green);
}

void CMainFrame::OnUpdateBlueUI(CCmdUI* pCmdUI)
{
    // Press or unpress the toolbar button.
    pCmdUI->SetCheck(m_color == Blue);
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

You've now completed the second version of the Toolbar application. Choose Developer Studio's Build, Build command to compile and link the application. Then, choose Build, Execute (or just press Ctrl+F5) to run the application.

## Creating Custom Toolbars

Normally, the toolbars that you create with MFC support only button controls. You know from using other Windows applications, however, that a toolbar can display many types of controls. For example, Word for Windows creates toolbars that contain several drop-down lists for selecting things like fonts and paragraph styles. You can create such custom toolbars by writing your own toolbar class. This is easier than you think, because you can derive the new class from MFC's CToolBar class, and then just add the extra features to the class that you need.

The third version of the Toolbar application demonstrates this powerful programming technique. Perform the following steps to modify the Toolbar application so that it uses a custom toolbar.



The complete source code and executable file for the Toolbar3 application can be found in the Chap11\Toolbar3 directory on this book's CD-ROM.

1. Copy the files MyToolbar.h and MyToolbar.cpp to the Toolbar application's project directory.
2. Use Developer Studio's Project, Add to Project, Files command to add the MyToolbar.cpp file to the project's other files.
3. Load mainfrm.h, and add the following line near the top of the file, right after the line #include <afxext.h>:

```
#include "mytoolbar.h"
```

4. While still in mainfrm.h, change the line CToolBar m\_toolbar to the following:

```
CMyToolbar m_toolbar;
```

5. Add the following line to mainfrm.h, right after the line int m\_color:

```
CString m_displayStr;
```

6. Add the following lines to mainfrm.h right after the line virtual BOOL PreCreateWindow(CREATESTRUCT& cs):

```
// Local member functions.
public:
    void ShowSelection(LPCSTR str);
```

7. Load `mainfrm.cpp`, and add the following lines to the end of the class's constructor:

```
// Set the display string.  
m_displayStr = "Item 1";
```

8. Replace all the code in the body of the class's `OnCreate()` function with the lines shown in Listing 11.10.

---

**Listing 11.10** `lst11_10.cpp`—New Code for the Window's `OnCreate()` Function

---

```
// Call the base class's OnCreate().  
CFrameWnd::OnCreate(lpCreateStruct);  
  
// Create and load the toolbar.  
m_toolbar.Create(this);  
  
// Enable the window for docking.  
EnableDocking(CBRS_ALIGN_ANY);  
DockControlBar(&m_toolbar);  
  
return 0;
```

---

9. Add the following lines to the end of the `OnPaint()` function:

```
// Show the display string.  
paintDC.TextOut(152, 150, m_displayStr);
```

10. Add the function shown in Listing 11.11 to the end of the `mainfrm.cpp` file.

---

**Listing 11.11** `lst11_11.cpp`—The New `ShowSelection()` Member Function

---

```
void CMainFrame::ShowSelection(LPCSTR str)  
{  
    m_displayStr = str;  
    Invalidate();  
}
```

---

11. Start Developer Studio's toolbar editor, and add a placeholder button to the toolbar, as shown in Figure 11.12. (The placeholder button can have any ID, and you can draw anything as the button's icon.) The placeholder button reserves space for the combo box control.





**FIG. 11.12** You need to add a placeholder button to the toolbar.

You've now completed the third version of the Toolbar application. Choose Developer Studio's **B**uild, **B**uild command to compile and link the application. Then, choose **B**uild, **E**xecute (or just press Ctrl+F5) to run the application. When you do, you see the window shown in Figure 11.13. The window not only displays a rectangle of the selected color, but also the string that's currently selected in the toolbar's combo box. To change the displayed string, just select a new item in the combo box.



**FIG. 11.13** The third version of the Toolbar application features a custom toolbar.

## Exploring the Custom Toolbar Class

The toolbar displayed in the new version of the toolbar application is very much like the standard MFC toolbar you've been learning about in this chapter. The only difference is that the new toolbar includes a combo box, as well as buttons. How is this bit of MFC magic performed? Listings 11.12 and 11.13 show the custom toolbar class's header and implementation files.

### Listing 11.12 MyToolbar.h—The CMyToolbar Class's Header File

```

////////////////////////////////////
// MYTOOLBAR.H: Header file for the CMyToolbar class,
//               which represents the application's custom
//               toolbar.
////////////////////////////////////

#ifndef __MYTOOLBAR_H
#define __MYTOOLBAR_H

#include <afxext.h>

class CMyToolbar : public CToolBar
{
// Protected data members.
protected:
    CComboBox m_comboBox;

// Constructor and destructor.
public:

```

```
CMyToolbar();
~CMyToolbar();

// Message map functions.
public:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnComboBox();

    DECLARE_MESSAGE_MAP()
};

#endif
```

---

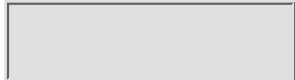
<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 + [Advanced Search](#)  
 + [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The program uses the returned index to get the string that represents the selected item, like this:

```
char str[25];
m_comboBox.GetLBText(index, str);
```

Finally, the program gets a pointer to the main window and calls the window’s local ShowSelection() member function through that pointer:

```
CMainFrame* mainWindow = (CMainFrame*)AfxGetMainWnd();
mainWindow->ShowSelection(str);
```

The main window’s ShowSelection() member function sets the window’s display string (m\_displayStr) to the selected string and calls Invalidate() to force the window to redraw its display with the new data.

## Working with Status Bars

Status bars are mostly benign objects that sit at the bottom of your application’s window, doing whatever MFC instructs them to do. This consists of displaying command descriptions and showing the status of various keys on the keyboard, including the Caps Lock and Scroll Lock keys. In fact, status bars are so mundane from the programmer’s point of view that they aren’t even represented by a resource that you can edit like a toolbar.

Still, a status bar, just like a toolbar, must reflect the interface needs of your specific application. For that reason, MFC’s CStatusBar class features a set of methods with which you can customize the status bar’s appearance and operation. Table 11.2 lists the methods along with brief descriptions.

**Table 11.2 Methods of the CStatusBar class**

Method	Description
CommandToIndex()	Gets an indicator’s index, given its ID
Create()	Creates the status bar
GetItemID()	Gets an indicator’s ID, given its index
GetItemRect()	Gets an item’s display rectangle, given its index
GetPaneInfo()	Gets information about an indicator
GetPaneStyle()	Gets an indicator’s style

GetPaneText()	Gets an indicator's text
GetStatusBarCtrl()	Gets a reference to the CStatusBarCtrl object represented by the CStatusBar object
SetIndicators()	Sets the indicators' IDs
SetPaneInfo()	Sets the indicators' ID, width, and style
SetPaneStyle()	Sets an indicator's style
SetPaneText()	Sets an indicator's text

---

When you create a status bar as part of an AppWizard application, a window similar to that shown in Figure 11.14 appears. The status bar has several parts, called panes, that display certain information about the status of the application and the system. These panes, which are marked in Figure 11.14, include indicators for the Caps Lock, Num Lock, and Scroll Lock keys, as well as a message area for showing status text and command descriptions. To see a command description, place your mouse pointer over a button on the toolbar (see Figure 11.15).



**FIG. 11.14** The default MFC status bar contains a number of informative panes.



**FIG. 11.15** The message area is used mainly for command descriptions.

Currently, the Toolbar application you've been building in this chapter features no status bar. Although Toolbar is not an AppWizard generated program, you can still add a status bar. Perform the following steps to add a status bar to the application.



The complete source code and executable file for the Toolbar4 application can be found in the Chap11\Toolbar4 directory on this book's CD-ROM.

1. Load mainfrm.h, and add the following line to the class's data member declarations, right after the line `CString m_displayStr`, which you placed there previously.

```
CStatusBar m_statusBar;
```

2. Load mainfrm.cpp, and add the lines in Listing 11.15 right after the class's message map:

#### **Listing 11.15** lst11\_15.cpp—Array of Status Bar Indicators

---

```
static UINT indicators[] =
{
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

---

**3.** Add the following lines near the end of the OnCreate() function, right before the line return 0:

```
// Create the status bar.
m_statusBar.Create(this);
m_statusBar.SetIndicators(indicators, 4);
```

**4.** Use the string table editor to add three strings to the application's string table. The three string IDs should be ID\_INDICATOR\_CAPS, ID\_INDICATOR\_NUM, and ID\_INDICATOR\_SCRL. The three strings associated with the IDs should be "CAPS", "NUM", and "SCRL", respectively, as shown in Figure 11.16.



**FIG. 11.16** Adding indicator strings to the application's string table.

You've now completed the fourth version of the Toolbar application. Choose Developer Studio's **B**uild, **B**uild command to compile and link the application. Then, choose **B**uild, **E**xecute (or just press Ctrl+F5) to run the application. When you do, the window shown in Figure 11.17 appears. The window now boasts a nifty status bar. Press your keyboard's Caps Lock, Scroll Lock, and Num Lock keys to see the status bar's indicators turn on and off. Also, place your mouse pointer over one of the toolbar buttons, and its description appears in the status bar's message area.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced](#)  
[Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Understanding Status Bar Basics

The status bar that you created in the previous section is identical to the one that AppWizard creates for you when you use AppWizard to generate an application. Still, it's a snap to add to your application. The first thing you must do is decide what types of panes you want to include in the status bar. The standard status bar has panes for showing the status of the Caps Lock, Num Lock, and Scroll Lock keys, as well as a separator pane that's used to display command descriptions and other types of messages.



**FIG. 11.17** The sample application now has a status bar.

After you've determined the types of panes that you want to include in the status bar, you must create an array that contains IDs for each of the panes, as shown in Listing 11.16:

### Listing 11.16 lst11\_16.cpp—Creating an Array of Indicator IDs

```
static UINT indicators[] =
{
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

MFC defines a set of default IDs that automatically link certain keyboard keys to status indicators as well as define separator panels. Listing 11.16 uses four of those predefined IDs in the indicator array. Other indicator IDs you can use are ID\_INDICATOR\_OVR, ID\_INDICATOR\_KANA, ID\_INDICATOR\_EXT, and ID\_INDICATOR\_REC.

After creating the indicator array, you can create the status bar. You do this in the main window's OnCreate() function, where you also create the window's toolbar:

```
m_statusBar.Create(this);
```

The status bar's `Create()` function takes a pointer to the parent window as its argument. You can also specify styles when you call `Create()`. You'll rarely need to do this, however, because the default styles work just fine for virtually every status bar.

After you create the status bar, you define its panes by calling the status bar object's `SetIndicators()` member function, like this:

```
m_statusBar.SetIndicators(indicators, 4);
```

The `SetIndicators()` function takes two arguments, which are the name of the ID array and the number of elements in the array.

## Creating Custom Panes

If you use the predefined IDs for your status bar's panes, you need do nothing more than specify the IDs in the ID array. When the status bar appears in your application's window, MFC takes care of the rest. However, you might sometimes want to add your own panes to a status bar. To add a custom pane to a status bar, you must complete these steps:

1. Create a command ID for the new pane.
2. Create a default string for the pane.
3. Add the pane's command ID to the status bar's indicators array.
4. Create a command-update handler for the pane.

The following sections cover these steps in detail.

**Creating a New Command ID.** This step is easy, thanks to Developer Studio's symbol browser. To add the command ID, first choose the View, Resource Symbols command on Developer Studio's menu bar. When you do, you see the Resource Symbols dialog box (see Figure 11.18), which displays the currently defined symbols for your application's resources. Click the New button, and the New Symbol dialog box appears. Type the new ID into the Name box, and the ID's value into the Value box (see Figure 11.19). Usually, you can just accept the value that MFC suggests for the ID.



**FIG. 11.18** Use the Resource Symbols dialog box to add new command IDs to your application.



**FIG. 11.19** Type the new ID's name and value into the New Symbol dialog box.

Click the OK and Close buttons to finalize your selections, and your new command ID is defined.

**Creating the Default String.** The Visual C++ compiler insists that every status bar pane has a default string defined for it. To define a default string, first go to the ResourceView window (by clicking the ResourceView tab) and open the String Table resource into the string-table editor. Open the string-table editor by double-clicking the String Table resource.

Now, double-click the blank line in the string-table editor, which brings up the String Properties dialog box. Type the new pane's command ID into the ID box and the default string into the Caption box (see Figure 11.20). (Instead of typing the command ID, you can choose it from the drop-down list.)



**FIG. 11.20** Use the String Properties dialog box to define the new pane's default string.

**Adding the ID to the Indicators Array.** When MFC constructs your status bar, it uses an array of IDs to determine which panes to display and where to display them. As you now know, this array of IDs is passed as an argument to the status bar's `SetIndicators()` member function, which is called in the `CMainFrame` class's `OnCreate()` function.

To add your new pane to the array, type the pane's ID into the array at the position in which you want the new pane to appear in the status bar, followed by a comma. (The first pane, `ID_SEPARATOR`, should always remain in the first position.) Listing 11.17 shows the indicator array with the new pane added.

**Listing 11.17** `lst11_17.cpp`—Adding the New Pane's ID to the Indicator Array

---

```
static UINT indicators[] =
{
    ID_SEPARATOR,
    ID_CUSTOM_PANE,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

---

**Creating the Pane's Command Update Handler.** MFC does not automatically enable new panes when it creates the status bar. Instead, you



must create a command update handler for the new pane, and enable the pane yourself. You must also add whatever code is needed to display information in the pane, assuming that the default string you defined in an earlier step is only a placeholder.

- **See** “Writing Update-Command-UI Functions,” for more information on command-update handlers, **p. 113**.

First, you must declare the new command update handler in the mainfrm.h header file. The prototype you need to add to the header file’s message map functions looks like this:

```
afx_msg void OnCustomPane(CCmdUI *pCmdUI);
```

Of course, the actual name of the handler will vary from pane to pane, but the rest of the line should look exactly as it does here.

Next, you have to add the handler to the class’s message map (in mainfrm.cpp), which is what associates the command ID with the handler:

```
ON_UPDATE_COMMAND_UI(ID_CUSTOM_PANE, OnCustomPane)
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief
 Full

• [Advanced Search](#)

• [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

As you learned in Chapter 6, “Using Menus,” you use the ON\_UPDATE\_COMMAND\_UI macro to associate command IDs with their UI handlers. The macro’s two arguments are the command ID and the name of the command update handler.

Now, you’re ready to write the new command update handler. In the handler, you have to enable the new pane, as well as set the pane’s contents. Listing 11.18 shows the command update handler for the new pane:

**Listing 11.18** `lst11_18.cpp`—The Custom Pane’s Update Handler

```
void CMainFrame::OnCustomPane(CCmdUI *pCmdUI)
{
    pCmdUI->Enable();
    pCmdUI->SetText("This is a test");
}
```

If you don’t understand how a command update handler works, please refer to Chapter 6, “Using Menus,” where these important functions that control a command item’s appearance are discussed in detail. (No, a status bar pane is not a command item, but it uses the same mechanism for updating.)

**Modifying a Panel’s Appearance**

The panels that comprise a status bar have style settings just like most other Windows graphical elements. You can manipulate the appearance of a panel by using the status bar object’s member functions. The most useful is the SetPanelInfo() member function, which enables you to set the panel’s ID, style, and width. A call to SetPanelInfo() looks like this:

```
statusBar.SetPaneInfo(index, id, style, width);
```

The index argument is the zero-based index of the pane whose info you want to set. (The first pane has an index of 0, the second an index of 1, and so on.). The id argument is the panel’s ID, whereas style is the style flag for the panel, and width is the panel’s width. There are several styles you can use with a status bar panel, which are shown in Table 11.3.

**Table 11.3** Status Bar Panel Styles

Style	Description
SBPS_DISABLED	The panel does not display its contents.
SBPS_NOBORDERS	The panel has no 3D border.
SBPS_NORMAL	The panel has the default style.

SBPS\_POPOUT

The panel appears to pop out from the background rather than being recessed.

SBPS\_STRETCH

The pane stretches to fill unused space.

---

## Creating the Final Version of the Toolbar Application

If you want to see the status bar tricks described in the previous sections in action, perform the following steps to modify the Toolbar application.



The complete source code and executable file for the Toolbar5 application can be found in the Chap11\Toolbar5 directory on this book's CD-ROM.

1. Choose the View, Resource Symbols command on Developer Studio's menu bar. When you do, you see the Resource Symbols dialog box.
2. Click the New button, and the New Symbol dialog box appears.
3. Type the **ID\_CUSTOM\_PANE** into the Name box, and click OK. Click Close to finalize your changes.
4. Click the ResourceView tab to go the ResourceView window, and open the String Table resource into the string table editor.
5. Double-click the blank line in the string-table editor. The String Properties dialog box appears.
6. Type **ID\_CUSTOM\_PANE** into the ID box and **Default String** into the Caption box. Press Enter to finalize your changes.
7. Load mainfrm.h, and add the following line to the message map function declarations, right after the line `afx_msg void OnUpdateBlueUI(CCmdUI* pCmdUI)`, which you placed there previously:

```
afx_msg void OnCustomPane(CCmdUI *pCmdUI);
```

8. Load mainfrm.cpp, and add the following line to the class's message map, right after the line `ON_UPDATE_COMMAND_UI(ID_BLUE, OnUpdateBlueUI)`, which you placed there previously:

```
ON_UPDATE_COMMAND_UI(ID_CUSTOM_PANE, OnCustomPane)
```

9. Replace the class's indicator-ID array with the array shown in Listing 11.19.

### Listing 11.19 lst11\_19.cpp—The New Indicator Array

---

```
static UINT indicators[] =
{
    ID_SEPARATOR,
    ID_CUSTOM_PANE,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

---

10. In the `OnCreate()` function, change the 4 in the line `m_statusBar.SetIndicators(indicators, 4)` to a 5.

**11.** Add the following line near the end of OnCreate(), right before the return statement:

```
m_statusBar.SetPaneInfo(1, ID_CUSTOM_PANE, SBPS_POPOUT, 30);
```

**12.** Add the function shown in Listing 11.20 to the end of the mainfrm.cpp file:

### Listing 11.20 lst11\_20.cpp—The New Pane’s UI Handler

---

```
void CMainFrame::OnCustomPane(CCmdUI *pCmdUI)
{
    pCmdUI->Enable();
    pCmdUI->SetText(m_displayStr);
}
```

---

You’ve now completed the final version of the Toolbar application. Choose Developer Studio’s **B**uild, **B**uild command to compile and link the program. Then, choose **B**uild, **E**xecute (or just press Ctrl+F5) to run the application. When you do, you see the window shown in Figure 11.21. The window’s status bar now displays a custom pane, which shows the currently selected combo box string. To change the contents of the custom pane, choose a new string from the toolbar’s combo box.



**Fig. 11.21** The Toolbar application’s status bar now has a custom pane.

You now know everything you need to know to create almost any type of toolbar or status bar. Still, feel free to experiment with different style flags and settings when you create your toolbars and status bars. Although you should try to keep your window adornments looking as they do in most Windows applications, keeping them familiar to the user, you might come up with something useful for special situations. In the next chapter, “Property Sheets and Wizards,” you learn about property sheets, which are another important element of the modern Windows application.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb’s [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 12

# Property Sheets and Wizards

- How to create property page resources
- How to associate property sheets and pages with their MFC classes
- How to initialize and display property sheets
- How to convert a property sheet to a wizard
- How to respond to wizard buttons

One of the newest types of graphical objects is the **tabbed dialog box**, also known as a **property sheet**. Windows 95 is loaded with property sheets, which organize the many options that can be modified by the user. What's a property sheet? Basically, it's a dialog box with two or more pages. You flip the pages by clicking labeled tabs located at the top of the dialog box. By using such dialog boxes to organize complex groups of options, Windows 95 lets users more easily find the information and settings that they need. As you've probably guessed, MFC supports the Windows 95 property sheets, with the classes `CPropertySheet` and `CPropertyPage`.

Similar to property sheets are wizards, which use buttons to move from one page to another rather than using tabs. You've seen a lot of wizards, too. These special types of dialog boxes guide the user step by step through complicated processes. For example, when you use AppWizard to generate source code for a new MFC project, the wizard guides you through the entire process. To control the wizard, you click buttons labeled Back, Next, and Finish.

## Introducing Property Sheets

Finding a sample property sheet in Windows 95 is as easy as finding sand at the beach. Just click virtually any Properties command or double-click an icon in the Control Panel. For example, Figure 12.1 shows the dialog box that you see when you double-click the Control Panel's Add/Remove Programs icon. This is a property sheet that contains three pages labeled Install/Uninstall, Windows Setup, and Startup Disk, each page containing commands and options related to the page's title topic.

In Figure 12.1, you can see programs installed on the machine that Windows can automatically uninstall. There's also an Install button that leads to other dialog boxes that help you install new programs from floppy disks or

CD-ROMs. On the other hand, the Windows Setup page (see Figure 12.2) helps you add or remove files from the Windows system. To get to this page, you need only click the Windows Setup tab. The Startup Disk page, of course, houses yet another set of options.



**FIG. 12.1** The Add/Remove Programs Properties dialog box contains three tabbed pages.



**FIG. 12.2** To move to the Windows Setup page, you click the Windows Setup tab.

As you can see, property sheets are a great way to organize many types of related options. Gone are the days of dialog boxes so jam-packed with options that you needed a college-level course just to figure them out. In the sections that follow, you will learn to program your own tabbed property sheets using MFC's CPropertySheet and CPropertyPage classes.

## Creating the Property Sheet Demo Application

Now that you've had an introduction to property sheets, it's time to learn how to build an application that uses these handy, specialized dialog boxes. In the steps that come later, you'll build the Property Sheet Demo application, which demonstrates the creation and manipulation of property sheets. Follow the steps that come next to create the basic application and modify its resources.

### Creating the Basic Property Sheet Demo Application

The complete source code and executable file for this part of the Property Sheet Demo application can be found in the CHAP12\Psht, Part 1 directory of this book's CD-ROM.



1. Use AppWizard to create the basic files for the Property Sheet Demo program, selecting the options listed in the following table. When you're done, the New Project Information dialog box appears; it should look like Figure 12.3. Click the OK button to create the project files.

---

Dialog Box Name	Options to Select
-----------------	-------------------

---

New Project	Name the project psht and then set the project path to the directory into which you want to store the project's files. Leave the other options set to their defaults.
Step 1	Select Single Document.
Step 2 of 6	Leave set to defaults.
Step 3 of 6	Leave set to defaults.
Step 4 of 6	Turn off all application features.
Step 5 of 6	Leave set to defaults.
Step 6 of 6	Leave set to defaults.

---



**FIG. 12.3** Your New Project Information dialog box should look like this.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

2. Select the ResourceView tab in the project workspace window. Visual C++ displays the ResourceView window, shown in Figure 12.4.
3. In the ResourceView window, click the plus sign next to psht resources to display the application's resources. Click the plus sign next to Menu and then double-click the IDR\_MAINFRAME menu ID. Developer Studio's menu editor appears.
4. Click the Property Sheet Demo application's Edit menu (not Developer Studio's Edit menu) and then press your keyboard's Delete key to delete the Edit menu. When you do, a dialog box asks for verification of the delete command. Click the OK button.



**FIG. 12.4** The ResourceView tab displays the ResourceView window.

5. Double-click the About psht& item in the Help menu and change it to **&About Property Sheet Demo**
6. In the application's File menu, delete all menu items except Exit.
7. Add a command called Property Sheet to the File menu, giving it the command ID ID\_PROPSHEET, as shown in Figure 12.5. Then use your mouse to drag the new command above the Exit command, so that it's the first command in the File menu.



**FIG. 12.5** Add a Property Sheet command to the File menu.

8. Double-click the Accelerator resource in the ResourceView window and highlight the IDR\_MAINFRAME accelerator ID. Press your Delete key to delete all accelerators from the application.
9. Double-click the Dialog resource in the ResourceView window. Double-click the IDD\_ABOUTBOX dialog box ID to bring up the dialog box editor.
10. Modify the dialog box by changing the title to **About Property Sheet Demo**; changing the first static text string to **Property Sheet Demo, Version 1.0**; and adding the static string **by Macmillan**



**Computer Publishing**, as shown in Figure 12.6. Close the dialog box editor.



**FIG. 12.6** The About dialog box should look like this.

**11.** Double-click the String Table resource in the ResourceView window. Double-click the String Table ID to bring up the string-table editor.

**12.** Double-click the IDR\_MAINFRAME string and then change the first segment of the string to **Property Sheet Demo**, as shown in Figure 12.7. Close the string-table editor.



**FIG. 12.7** The first segment of the IDR\_MAINFRAME string appears in your main window's title bar.

You have now completed the first part of the Property Sheet Demo application. Compile and link the application by selecting the Build button in the toolbar, by selecting Developer Studio's Build, Build command, or by pressing F7. After building the application, you can run it if you like, but it won't do much until you've added more features, which you'll do in the following section.

## Creating Resources for Property Pages

Now that you have the application's basic resources the way you want them, it's time to add the resources that define the application's property sheet. This means creating dialog box resources for each page in the property sheet. Follow the next steps to complete this task.



The complete source code and executable file for this part of the Property Sheet Demo application can be found in the CHAP12\Psht, Part 2 directory of this book's CD-ROM.

**1.** Select the Insert, Resource command in Developer Studio's menu bar, and then double-click Dialog in the Insert resource dialog box to create a new dialog box resource. The new dialog box appears in the dialog box editor.

This dialog box, when properly set up, represents the first page of the property sheet.

**2.** Delete the OK and Cancel buttons by selecting each with your mouse and then pressing your keyboard's Delete key.

**3.** Select the dialog box and press Enter to bring up its Dialog Properties sheet. Change the ID to **IDD\_PAGE1DLG** and the caption to

**Page 1**, as shown in Figure 12.8.



**FIG. 12.8** Double-clicking the dialog box brings up its properties.

4. Click the Styles tab. Change the Syle box to Child, the Border box to Thin, and turn off the System Menu check box, as shown in Figure 12.9.

The Child style is necessary because the property page will be a child window of the property sheet. The property sheet itself provides the container for the property pages. (Actually, if you forget to set these styles, MFC seems to be smart enough to display the property page correctly, in spite of your oversight.)



**FIG. 12.9** A property page uses different styles than those used in a regular dialog box.

5. Add an edit box to the property page, as shown in Figure 12.10.

6. Create a second property page by following the previous steps 1 through 5. For this property page, use the ID **IDD\_PAGE2DLG**, a caption of **Page 2**, and add a check box rather than an edit control, as shown in Figure 12.11.



**FIG. 12.10** A property page can hold whatever controls you want.



**FIG. 12.11** The second property page should look like this.

You have now completed the second part of the Property Sheet Demo application. If you want, you can compile and link the application by selecting the Build button in the toolbar, by selecting Developer Studio's Build, Build command, or by pressing F7. However, before the application will do anything interesting, you need to create classes for your new property pages.

## Creating Classes for Property Pages and Property Sheets

You now have all your resources created. However, you need to associate your two new property page resources with C++ classes so that you can control them in your program. You also need a class for your property sheet, which will hold the property pages that you've created. Follow the steps given next to create the new classes.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)



The complete source code and executable file for this part of the Property Sheet Demo application can be found in the CHAP12\Psht, Part 3 directory of this book's CD-ROM.

1. Make sure that the Page 1 property page is visible in the dialog edit box and then either click the Developer Studio's ClassWizard button or select View, ClassWizard from the menu bar. The MFC ClassWizard property sheet appears, displaying the Adding a Class dialog box.
2. Select the Create New Class option and then click the OK button. The Create New Class dialog box appears.
3. In the Name box, type **CPage1**, and in the Base Class box, select CPropertyPage. Then click the OK button to create the class. You've now associated the property page with an object of the CPropertyPage class, which means that you can use the object to manipulate the property page as needed. The CPropertyPage class will be especially important when you learn about wizards.
4. Select the Member Variables tab of the MFC ClassWizard property sheet. With IDC\_EDIT1 highlighted, click the Add Variable button. The Add Member Variable dialog box appears.
5. Name the new member variable **m\_edit**, as shown in Figure 12.12, and then click the OK button. ClassWizard adds the member variable to the new CPage1 class. The member variable will hold the value of the property page's control.



**FIG. 12.12** ClassWizard makes it easy to add member variables.

6. Click OK on the MFC ClassWizard Properties sheet to finalize the creation of the CPage1 class.
7. Follow steps 1 through 7 for the second property sheet. Name the class **CPage2** and add a Boolean member variable called **m\_check** for the IDC\_CHECK1 control, as shown in Figure 12.13.



**FIG. 12.13** The second property page needs a Boolean member variable called `m_check`.

8. Close all the resource editor windows. Then either press Ctrl+W or select View, ClassWizard from the menu bar. The MFC ClassWizard Properties sheet reappears.

9. Select New from the Add Class menu button. The New Class dialog box appears.

10. In the Name box, type **CPropSht**, select CPropertySheet in the Base Class box, and then click the OK button. ClassWizard creates the CPropSht class. Click the MFC ClassWizard Properties sheet's OK button to finalize the class.

In the finished application, you'll use the new CPropSht class to create and manipulate the property sheet.

You have now completed the third part of the Property Sheet Demo application. If you want, you can compile and link the application by selecting the Build button in the toolbar, by selecting Developer Studio's Build, Build command, or by pressing F7. You still have a little work to do, though, adding the program lines that complete the application. You'll do that in the next section.

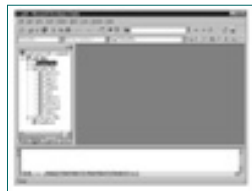
## Completing the Application

At this point, you have three new classes—CPage1, CPage2, and CPropSht—in your program. The first two classes are derived from MFC's CPropertyPage class, and the third is derived from CPropertySheet. Although ClassWizard has created the basic source-code files for these new classes, you still have to add code to the classes to make them work the way you want. Follow the next set of steps to complete the Property Sheet Demo application.

The complete source code and executable file for this final part of the Property Sheet Demo application can be found in the CHAP12\Psht directory of this book's CD-ROM.



1. Click the FileView tab in order to display the FileView window. Then display the project's header files by clicking the plus sign next to the folder labeled "psht files," and double-clicking the Header Files folder, as shown in Figure 12.14.



**FIG. 12.14** The FileView window lists the source files that make up your project.

2. Double-click the PropSht.h entry in the file list. The PropSht.h file appears in the code window.

3. Add the following lines near the top of the file, right before the beginning of the class's declaration:

```
#include "page1.h"
#include "page2.h"
```

These lines give the CPropSht class access to the CPage1 and CPage2 classes, so that the property sheet can declare member variables of these property page classes.

**4.** Add the following lines to the CPropSht class's Attributes section, right after the public keyword:

```
CPage1 m_page1;
CPage2 m_page2;
```

These lines declare the class's data members, which are the property pages that'll be displayed in the property sheet.

**5.** Close the PropSht.h file (saving the changes), and then load the PropSht.cpp source file. Add the following lines to the class's second constructor:

```
AddPage( &m_page1 );
AddPage( &m_page2 );
```

The preceding lines add the two property pages to the property sheet when the sheet is constructed.

**6.** Open the pshtView.h file. Add the following lines to the class's Attributes section, right after the line CPshtDoc\* GetDocument():

```
protected:
    CString m_edit;
    BOOL m_check;
```

These lines declare two data members for the view class. These data members will hold the selections made in the property sheet by the user.

**7.** Close the pshtView.h file (saving the changes), and load the pshtView.cpp file. Add the following line near the top of the file, after the #endif compiler directive:

```
#include "propsht.h"
```

This line gives the view class access to the CPropSht class, so that it can create the property sheet when requested to do so.

**8.** Add the following lines to the view class's constructor:

```
m_edit = "Default";
m_check = FALSE;
```

These lines initialize the class's data members so that, when the

property sheet appears, these default values can be copied into the property sheet's controls. After the user changes the contents of the property sheet, these data members will always hold the last values from the property sheet, so those values can be restored to the sheet when needed.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

9. Add the lines shown in Listing 12.1 to the OnDraw() function, right after the TODO: Add draw code for native data here comment.

#### Listing 12.1 LST12\_01.CPP—Code for the OnDraw() Function

```

pDC->TextOut(20, 20, m_edit);
if (m_check)
    pDC->TextOut(20, 50, "TRUE");
else
    pDC->TextOut(20, 50, "FALSE");
  
```

These lines display the current selections from the property sheet. At the start of the program, the default values are displayed.

10. Press Ctrl+W or select the **View, ClassWizard** command from the menu bar. The ClassWizard property sheet appears.

11. Click the Message Maps tab, and make sure that CPshView is selected in the Class Name and Object IDs boxes. Then add the OnPropsheet() message response function, as shown in Figure 12.15.

The OnPropsheet() function is now associated with the Property Sheet command that you previously added to the File menu. That is, when the user selects the Property Sheet command, MFC calls OnPropsheet(), where you can respond to the command.



**FIG. 12.15** Use ClassWizard to add the OnPropsheet() member function.

12. Click the **Edit Code** button to jump to the OnPropsheet() function, and then add the lines shown in Listing 12.2 right after the TODO: Add your command handler code here comment.

#### Listing 12.2 LST12\_02.CPP—Code for the OnPropSheet() Function

```

CPropSht propSheet("Property Sheet", this, 0);
propSheet.m_page1.m_edit = m_edit;
propSheet.m_page2.m_check = m_check;
int result = propSheet.DoModal();
  
```



```
if (result == IDOK)
{
    m_edit = propSheet.m_page1.m_edit;
    m_check = propSheet.m_page2.m_check;
    Invalidate();
}
```

---

The code segment in Listing 12.2, which is discussed in more detail a little later in this chapter in the section entitled “Understanding the Property Sheet Demo Application,” creates, displays, and manages the property sheet.

You’ve now finished the complete application. Click the toolbar’s Build button—or select the Build, Build command from the menu bar—to compile and link the application.

## Running the Property Sheet Demo Application

Once you have the program compiled, run it. When you do, you see the window shown in Figure 12.16. As you can see, the window displays two values, which are the default values for the controls in the application’s property sheet. You can change these values using the property sheet. To do this, select the File menu’s Property Sheet command. The property sheet appears on the screen (see Figure 12.17). The property sheet contains two pages, each of which holds a single control. When you change the settings of these controls and click the property sheet’s OK button, the application’s window displays the new values.



**FIG. 12.16** When it starts, the Property Sheet Demo application displays default values for the property sheet’s controls.



**FIG. 12.17** The application’s property sheet contains two pages.

## Understanding the Property Sheet Demo Application

Previously, in the section titled “Creating the Property Sheet Demo Application,” you went through the process of creating Property Sheet Demo, step by step. During this process, you discovered that you must complete several tasks in order to add property sheets to your application. Each of those tasks was explained in the steps’ text. However, to give you a clearer picture of what you did, the steps that are most important to the creation of a property sheet are summarized here:

1. Create a dialog box resource for each page in the property sheet. These resources should have the Child and Thin styles and should have no system menu.
2. Associate each property page resource with an object of the CPropertyPage class. You can do this easily with ClassWizard.
3. Create a class for the property sheet, deriving the class from MFC's CPropertySheet class. You can generate this class using ClassWizard.
4. In the property sheet class, add member variables for each page that you'll be adding to the property sheet. These member variables must be instances of the property page classes that you created in Step 2.
5. In the property sheet's constructor, call AddPage() for each page in the property sheet.
6. To display the property sheet, call the property sheet's constructor and then call the property sheet's DoModal() member function, just as you would with a dialog box.

---

**Note:**

As you read over the steps required for creating a property sheet, be sure that you understand the difference between a property sheet and a property page. A *property sheet* is a window that contains property pages. *Property pages* are windows that hold the controls that will appear on the property sheet's pages.

---

After you have your application written and have defined the resources and classes that represent the property sheet (or sheets; you can have more than one), you need a way to enable the user to display the property sheet when it's needed. In Property Sheet Demo, this is done by associating a menu item with a message response function. However you handle the command to display the property sheet, though, the process of creating the property sheet is the same. First, you must call the property sheet class's constructor, which Property Sheet Demo does like this:

```
CPropSht propSheet("Property Sheet", this, 0);
```

Here, the program is creating an instance of the CPropSht class. This instance (or object) is called propSheet. The three arguments are the property sheet's title string, a pointer to the parent window (which, in this case, is the view window), and the zero-based index of the first page to display. Because the property pages are created in the property sheet's constructor, creating the property sheet also creates the property pages.

Once you have the property sheet object created, you can initialize the data members that hold the values of the property page's controls, which Property Sheet Demo does like this:

```
propSheet.m_page1.m_edit = m_edit;  
propSheet.m_page2.m_check = m_check;
```

Now it's time to display the property sheet on the screen, which you do just as if it were a dialog box, by calling the property sheet's DoModal() member function:

```
int result = propSheet.DoModal();
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

DoModal() doesn't take any arguments, but it does return a value indicating which button the user clicked to exit the property sheet. In the case of a property sheet or dialog box, you'll usually want to process the information entered into the controls only if the user clicked the OK button, which is indicated by a return value of IDOK. Listing 12.3 shows how the Property Sheet Demo application handles the return value.

### Listing 12.3 LST12\_03.CPP—Handling the Property Sheet's Return Value

```

if (result == IDOK)
{
    m_edit = propSheet.m_page1.m_edit;
    m_check = propSheet.m_page2.m_check;
    Invalidate();
}
  
```

In Listing 12.3, the program retrieves the values of the controls from the property pages and then calls Invalidate() to force the window to be redrawn. If the user exits the property sheet by clicking the Cancel button, the code in the body of the if statement is ignored, and the window is not updated.

## Changing Property Sheets to Wizards

When you come right down to it, a *wizard* is nothing more than a property sheet that uses Back, Next, and Finish buttons instead of tabs. Because of the lack of tabs, however, the user must switch from one page to another in sequence. This forced sequence makes wizards terrific for guiding your application's users through the steps needed to complete a complex task. You've already seen how AppWizard in Visual C++ makes it easy to start a new project. You can create your own wizards that are suited to whatever application you want to build. In the following sections, you'll see how easy it is to convert a property sheet to a wizard.

### Running the Wizard Demo Application

In the CHAP12\WIZ folder of this book's CD-ROM, you'll find the Wizard Demo application. This application was built in much the same way as the Property Sheet Demo application that you created earlier in this chapter. However, as you'll soon see, there are a few differences in the Wizard Demo application that enable the user to access and use the application's wizard.

When you run the Wizard Demo application, the main window appears, including a File menu from which you can select the Wizard command. The Wizard command brings up the wizard shown in Figure 12.18.



**FIG. 12.18** The Wizard Demo application displays a wizard rather than a property sheet.

The wizard isn't too fancy, but it does demonstrate what you need to know in order to program more

complex wizards. As you can see, this wizard has three pages. On the first page is an edit control and three buttons called Back, Next, and Cancel. The Back button is disabled, because there is no previous page to go back to. The Cancel button enables the user to dismiss the wizard at any time, canceling whatever process the wizard was guiding the user through. The Next button causes the next page in the wizard to be displayed.

You can change whatever is displayed in the edit control if you like. However, the magic really starts when you click the Next button, which displays Page 2 of the wizard, shown in Figure 12.19. Page 2 contains a check box and the Back, Next, and Cancel buttons. Now the Back button is enabled, so that you can return to Page 1 if you want. Go ahead and click the Back button. The wizard tells you that the check box must be checked, as shown in Figure 12.20. As you'll soon see, this feature of a wizard enables you to verify the contents of a specific page before letting the user advance to another step.



**FIG. 12.19** In Page 2 of the wizard, the Back button is enabled.



**FIG. 12.20** You must select the check box before the wizard lets you leave Page 2.

After selecting the check box, you can click the Back button to move back to Page 1 or click the Next button to advance to Page 3. If you advance to Page 3, you see the display shown in Figure 12.21. Here, the Next button has changed to the Finish button, because you are on the wizard's last page. If you click the Finish button, the program displays a message box, after which the wizard disappears.



**FIG. 12.21** This is the last page of the Wizard Demo application's wizard.

## Creating Wizard Pages

As far as your application's resources go, you create wizard pages exactly as you create property sheet pages: by creating dialog boxes and changing the dialog box styles. You also need to associate each page that you create with an object of the CPropertyPage class. However, in order to take control of the pages in your wizard and keep track of what the user is doing with the wizard, there are a few member functions in the CPropertyPage class that you can override in your property page classes. These functions are OnSetActive(), OnWizardBack(), OnWizardNext(), and OnWizardFinish(). Read on to see how to use these functions.

## Setting the Wizard's Buttons

MFC automatically calls the OnSetActive() member function immediately upon displaying a specific page of the wizard. For example, when the program displays Page 1 of the wizard, the CPage1 class's OnSetActive() function gets called. In the Wizard Demo application, the CPage1 class's version of OnSetActive() looks like Listing 12.4.

### Listing 12.4 LST12\_04.CPP—The OnSetActive() Member Function

---

```
BOOL CPage1::OnSetActive()  
{  
    // TODO: Add your specialized code here and/or call the base class  
    CPropertySheet* parent = (CPropertySheet*)GetParent();  
    parent->SetWizardButtons(PSWIZB_NEXT);  
    return CPropertyPage::OnSetActive();  
}
```

---

In Listing 12.4, the program first gets a pointer to the wizard's property sheet window, which is the page's parent window. Then the program calls the wizard's `SetWizardButtons()` function, which determines the state of the wizard's buttons. `SetWizardButtons()` takes a single argument, which is a set of flags indicating how the page should display its buttons. These flags are `PSWIZB_BACK`, `PSWIZB_NEXT`, `PSWIZB_FINISH`, and `PSWIZB_DISABLED_FINISH`. The button flags that you include will enable the associated button (except for the `PSWIZB_DISABLED_FINISH` flag, which disables the Finish button). Because the call to `SetWizardButtons()` in Listing 12.4 includes only the `PSWIZB_NEXT` flag, only the Next button in the page will be enabled.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



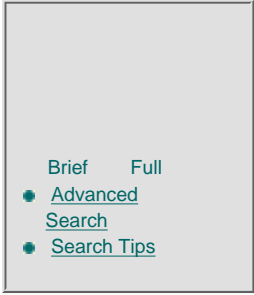
FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGEBROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Because the CPage2 class represents Page 2 of the wizard, its call to SetWizardButtons() enables both the Back and Next buttons, by ORing together the appropriate flags, like this:

```
parent->SetWizardButtons(PSWIZB_BACK | PSWIZB_NEXT);
```

Because Page 3 of the wizard is the last page, the CPage3 class calls SetWizardButtons(), like this:

```
parent->SetWizardButtons(PSWIZB_BACK | PSWIZB_FINISH);
```

This set of flags enables the Back button and changes the Next button to the Finish button.

## Responding to the Wizard's Buttons

In the simplest case, MFC takes care of everything that needs to be done in order to flip from one wizard page to the next. That is, when the user clicks a button, MFC springs into action and performs the Back, Next, Finish, or Cancel command. However, you'll often want to perform some action of your own when the user clicks a button. For example, you may want to verify that the information the user entered into the currently displayed page is correct. If there's a problem with the data, you can force the user to fix it before moving on.

To respond to the wizard's buttons, you can override the OnWizardBack(), OnWizardNext(), and OnWizardFinish() member functions. When the user clicks a wizard button, MFC calls the matching function in which you can do whatever is needed to process that page. An example is the way the wizard in the Wizard Demo application won't let you leave Page 2 until you've checked the check box. This is accomplished by overriding the functions shown in Listing 12.5.

### Listing 12.5 LST12\_05.CPP—Responding to Wizard Buttons

```
LRESULT CPage2::OnWizardBack()
{
    // TODO: Add your specialized code here and/or call the base class
    CButton *checkBox = (CButton*)GetDlgItem(IDC_CHECK1);
    if (!checkBox->GetCheck())
    {
        MessageBox("You must check the box.");
        return -1;
    }
    return CPropertyPage::OnWizardBack();
}

LRESULT CPage2::OnWizardNext()
{
    // TODO: Add your specialized code here and/or call the base class
    CButton *checkBox = (CButton*)GetDlgItem(IDC_CHECK1);
    if (!checkBox->GetCheck())
    {
        MessageBox("You must check the box.");
        return -1;
    }
}
```

```
    }  
    return CPropertyPage::OnWizardNext();  
}
```

---

In the functions in Listing 12.5, the program gets a pointer to the page's check box by calling the `GetDlgItem()` function. With the pointer in hand, the program can call the check box class's `GetCheck()` function, which returns a 1 if the check box is checked. If `GetCheck()` returns a 0, the program displays a message box and returns -1 from the function. Returning -1 tells MFC to ignore the button click and not change pages.

## Displaying a Wizard

As you've just learned, almost all the work involved in controlling a wizard is done in the classes that represent the wizard's pages. The property sheet class that represents the wizard works exactly the same way as it did in the property sheet example. However, there is one extra thing you must do when displaying a wizard, which is to call the property sheet's `SetWizardMode()` member function. This function call tells MFC that it should display the property sheet as a wizard rather than as a conventional property sheet. Listing 12.6 shows the view class's `OnWizard()` member function, which is the function that responds to the File menu's Wizard command.

### Listing 12.6 LST12\_06.CPP—Displaying a Property Sheet as a Wizard

---

```
void CWizView::OnWizard()  
{  
    // TODO: Add your command handler code here  
  
    CWizSheet wizSheet("Sample Wizard", this, 0);  
    wizSheet.m_page1.m_edit = m_edit;  
    wizSheet.m_page2.m_check = m_check;  
    wizSheet.SetWizardMode();  
    int result = wizSheet.DoModal();  
    if (result == ID_WIZFINISH)  
    {  
        m_edit = wizSheet.m_page1.m_edit;  
        m_check = wizSheet.m_page2.m_check;  
    }  
}
```

---

Notice in Listing 12.6 that the program creates the wizard almost exactly the same as a property sheet, the only difference being the call to `SetWizardMode()`. The wizard is displayed exactly the same as any other dialog box, by calling the object's `DoModal()` member function. There is, however, one difference in how you respond to the result returned by `DoModal()`. Because a wizard has no OK button, `DoModal()` cannot return `IDOK`. Instead, `DoModal()` returns `ID_WIZFINISH` if the user exits via the Finish button.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Chapter 13

# The Rich Edit Control

- Discover how to add a rich edit control to an application's window
- Learn to set a rich edit control's character attributes
- Experiment with paragraph-alignment styles
- Learn how to take advantage of MFC's powerful CRich EditView class

If you took all the energy that's been expended on writing text-editing software and concentrated that energy on other, less mundane programming problems, computer science would probably be a decade ahead of where it is now. Okay, that might be an exaggeration, but it is true that, when it comes to text editors, a huge amount of effort has been dedicated to reinventing the wheel. Wouldn't it be great to have one piece of text-editing code that all programmers could use as the starting point for their own custom text editors?

With Visual C++'s CRichEditCtrl control, which gives programmers access to Windows 95's *rich edit control*, you can get a huge jump on any text-editing functionality that you need to install in your applications. The rich edit control is capable of handling fonts, paragraph styles, and text color, as well as other types of tasks that are traditionally found in text editors. In fact, a rich edit control provides a solid starting point for any text-editing tasks that your application must handle.

## Introducing the Rich Edit Control

For all intents and purposes, you can consider the rich edit control to be a sort of "black box" word processor. That is, to add text-editing capabilities to your MFC application, you need to do very little programming. Nor do you need to know much about the inner workings of a rich edit control. Just place the control in your application's main window and call the control object's member functions to control it. A rich edit control enables the user to perform the following text-editing tasks:

- Type text
- Edit text using cut-and-paste and sophisticated drag-and-drop operations
- Set text attributes such as font, point-size, and color
- Apply underline, bold, italic, strikethrough, superscript, and subscript properties to text

- Format text using various alignments
- Lock text from further editing
- Save and load files

As the preceding list proves, a rich edit control is powerful. It is, in fact, almost a complete word-processor-in-a-box that you can plug into your program and use immediately. Of course, because a rich edit control offers so many features, there's a lot to learn. In this chapter, you'll get an introduction to creating and manipulating the rich edit control.

## Creating the RichEdit Application

There are a couple of ways you can add rich edit text-editing capabilities to your application using Visual C++. The first way is to add a rich edit control to your application's frame or view window. When you use this method, you are completely responsible for dealing with the control. Another way to add a rich edit control to your application is to make your view window an object of MFC's CRichEditView class. When you use this method, MFC handles much of the work for you.

In this section, you'll see how to directly create and manipulate a rich edit control. Using this method, you get an introduction to the control itself and how your MFC program manipulates it directly. Just follow these steps to create the RichEdit application.



The complete source code and executable file for the RichEdit application can be found in the CHAP13\RichEdit directory on this book's CD-ROM.

1. Start a new AppWizard project workspace called RichEdit, as shown in Figure 13.1.



**FIG.13.1** Here's how you should start the RichEdit project workspace.

2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 13.2.

Step 1: Single document

Step 2: Default settings

Step 3: Default settings

Step 4: Turn off Printing and Print Preview

Step 5: Default settings

Step 6: Default settings



**FIG. 13.2** These are the AppWizard settings for the RichEdit project.

3. Using the resource editor, remove the application's Edit menu. Also, remove all items from the File menu except for Exit.
4. Using the resource editor, remove all of the buttons from the application's toolbar.
5. Add four buttons to the application's toolbar, as shown in Figure 13.3. Give the buttons the IDs ID\_UNDERLINED, ID\_LEFT, ID\_CENTERED, and ID\_RIGHT.



**FIG. 13.3** Use the resource editor's painting tools to add four buttons to the application's toolbar.

6. Use ClassWizard to associate the ID\_UNDERLINED command with the OnUnderlined() message response function, as shown in Figure 13.4. Make sure that you have CMyRichEditView selected in the Class Name box before you add the function.



**FIG. 13.4** Add the OnUnderlined() message response function to the view class.

7. Click the Edit Code button, and then add the lines shown in Listing 13.1 to the new OnUnderlined() function, right after the TODO: Add your command handler code here comment:

**Listing 13.1 LST13\_01.CPP—Code for the OnUnderlined() Function**

---

```
CHARFORMAT charFormat;
charFormat.cbSize = sizeof(CHARFORMAT);
charFormat.dwMask = CFM_UNDERLINE;
m_richEdit.GetSelectionCharFormat(charFormat);

if (charFormat.dwEffects & CFE_UNDERLINE)
    charFormat.dwEffects = 0;
else
    charFormat.dwEffects = CFE_UNDERLINE;

m_richEdit.SetSelectionCharFormat(charFormat);
```

---

**8.** Use ClassWizard to associate the ID\_LEFT command with the OnLeft() message response function, as shown in Figure 13.5. Again, make sure that you have CMyRichEditView selected in the Class Name box before you add the function.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

9. Click the Edit Code button, and then add the lines shown in Listing 13.2 to the new OnLeft() function, right after the TODO: Add your command handler code here comment:

#### Listing 13.2 LST13\_02.CPP—Code for the OnLeft() Function

```

PARAFORMAT paraFormat;
paraFormat.cbSize = sizeof(PARAFORMAT);
paraFormat.dwMask = PFM_ALIGNMENT;
paraFormat.wAlignment = PFA_LEFT;
m_richEdit.SetParaFormat(paraFormat);
  
```



**FIG. 13.5** Add the OnLeft() message response function to the view class.

10. Use ClassWizard to associate the ID\_CENTERED command with the OnCentered() message response function in the view class, as shown in Figure 13.6.



**FIG. 13.6** Add the OnCentered() message response function to the view class.

11. Click the Edit Code button, and then add the lines shown in Listing 13.3 to the new OnCentered() function, right after the TODO: Add your command handler code here comment:

#### Listing 13.3 LST13\_03.CPP—Code for the OnCentered() Function

```

PARAFORMAT paraFormat;
paraFormat.cbSize = sizeof(PARAFORMAT);
paraFormat.dwMask = PFM_ALIGNMENT;
paraFormat.wAlignment = PFA_CENTER;
m_richEdit.SetParaFormat(paraFormat);
  
```

**12.** Use ClassWizard to associate the ID\_RIGHT command with the OnRight() message response function in the view class, as shown in Figure 13.7.



**FIG. 13.7** Add the OnRight() message response function to the view class.

**13.** Click the Edit Code button, and then add the lines shown in Listing 13.4 to the new OnRight() function, right after the TODO: Add your command handler code here comment:

**Listing 13.4 LST13\_04.CPP—Code for the OnRight() Function**

```
PARAFORMAT paraFormat;  
paraFormat.cbSize = sizeof(PARAFORMAT);  
paraFormat.dwMask = PFM_ALIGNMENT;  
paraFormat.wAlignment = PFA_RIGHT;  
m_richEdit.SetParaFormat(paraFormat);
```

**14.** Use ClassWizard to override the OnCreate() message response function in the view class, as shown in Figure 13.8.



**FIG. 13.8** Add the OnCreate() message response function to the view class.

**15.** Click the Edit Code button, and then add the following line to the new OnRight() function, right after the TODO: Add your command handler code here comment:

```
m_richEdit.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |  
    ES_AUTOVSCROLL | ES_MULTILINE,  
    CRect(0,0,388,302), this, 111);
```

**16.** Load the RichEditView.h header file, and add the following lines to the class's Attributes section, right after the line CMyRichEditDoc\* GetDocument() that's already there:

```
protected:  
    CRichEditCtrl m_richEdit;
```

**17.** Load the MainFrm.cpp file, and add the following lines to the beginning of the PreCreateWindow() function:

```
cs.cx = 400;
```

```
cs.cy = 400;
```

You've now completed the RichEdit application. To compile the application, choose Developer Studio's **Build, Build** command. You can then run the program by choosing the **Build, Execute** command. When you do, the application's main window appears. First, click in the window to give the rich edit control the focus. Then just start typing. Want to try out character attributes? Click the Underline button to add underlining to either selected text or the next text that you type. To try out paragraph formatting, click either the Left, Center, or Right buttons to specify paragraph alignment. Figure 13.9 shows the rich edit control with the different character and paragraph styles used.



**FIG. 13.9** A RichEdit control is a complete word processor.

---

**Note:**

If you carefully examine the RichEdit application's main window, you'll notice that you can see both the window's border and the rich edit control's border. The application shows the control's border so that you can see the control in the window. However, you can get rid of the edit control's border by removing the `WS_BORDER` flag from the edit control's window styles. When you do this, the main window will show no evidence of the control, except when you type text into it.

---

## Examining the RichEdit Application

Now that you've had a chance to try out a rich edit control, you'll want to examine the source code that accomplishes all those text-editing tricks. In the sections that follow, you'll see how to create a rich edit control and place it in a window. You'll also learn how to change default text settings such as character and paragraph formatting.

### Creating the Rich Edit Control

In the RichEdit application, the rich edit control is created in the `OnCreate()` member function, which MFC calls when the Windows sends the `WM_CREATE` message. In `OnCreate()`, the program creates the control by calling its `Create()` member function:

```
m_richEdit.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |  
    ES_AUTOVSCROLL | ES_MULTILINE,  
    CRect(0,0,388,302), this, 111);
```

This function's four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. The style constants include the same constants that you would use for creating any type of window, with the addition of special styles used with rich edit controls. Table 13.1 lists these special styles.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

**Table 13.1 Rich Edit Styles**

Style	Description
ES_AUTOHSCROLL	Automatically scrolls horizontally
ES_AUTOVSCROLL	Automatically scrolls vertically
ES_CENTER	Centers text
ES_LEFT	Left aligns text
ES_LOWERCASE	Lowercases all text
ES_MULTILINE	Enables multiple lines
ES_NOHIDESEL	Disallows hiding selected text when losing the focus
ES_OEMCONVERT	Converts from ANSI characters to OEM characters and back to ANSI
ES_PASSWORD	Displays characters as asterisks
ES_READONLY	Disables editing in the control
ES_RIGHT	Right aligns text
ES_UPPERCASE	Uppercases all text
ES_WANTRETURN	Inserts return characters into text when Enter is pressed

## Initializing and Manipulating the Rich Edit Control

As soon as the rich edit control is created, you might want to initialize it in some way. The `CRichEditCtrl` class features a number of member functions that enable you to initialize and manipulate the control. Those member functions and their descriptions are listed in Table 13.2.

**Table 13.2 Member Functions of the `CRichEditCtrl` Class**

Function	Description
<code>CanPaste()</code>	Determines whether the Clipboard's contents can be pasted into the control
<code>CanUndo()</code>	Determines whether the last edit can be undone
<code>Clear()</code>	Clears selected text
<code>Copy()</code>	Copies selected text to the Clipboard
<code>Create()</code>	Creates the control
<code>Cut()</code>	Cuts selected text to the Clipboard
<code>DisplayBand()</code>	Displays a portion of the control's text
<code>EmptyUndoBuffer()</code>	Resets the control's undo flag
<code>FindText()</code>	Finds the given text
<code>FormatRange()</code>	Formats text for an output target device
<code>GetCharPos()</code>	Gets the position of a given character
<code>GetDefaultCharFormat()</code>	Gets the default character format
<code>GetEventMask()</code>	Gets the control's event mask

GetFirstVisibleLine()	Gets the index of the first visible line
GetIRichEditOle()	Gets the IRichEditOle interface pointer for the control
GetLimitText()	Gets the maximum number of characters that can be entered
GetLine()	Gets the specified text line
GetLineCount()	Gets the number of lines in the control
GetModify()	Determines whether the control's contents have changed since the last save
GetParaFormat()	Gets the paragraph format of selected text
GetRect()	Gets the control's formatting rectangle
GetSel()	Gets the position of the currently selected text
GetSelectionCharFormat()	Gets the character format of selected text
GetSelectionType()	Gets the selected text's contents type
GetSelText()	Gets the currently selected text
GetTextLength()	Gets the length of the control's text
HideSelection()	Hides or shows selected text
LimitText()	Sets the maximum number of characters that can be entered
LineFromChar()	Gets the number of the line containing the given character
LineIndex()	Gets the character index of a given line
LineLength()	Gets the length of the given line
LineScroll()	Scrolls the text the given number of lines and characters
Paste()	Pastes the Clipboard's contents into the control
PasteSpecial()	Pastes the Clipboard's contents using the given format
ReplaceSel()	Replaces selected text with the given text
RequestResize()	Forces the control to send EN_REQUESTRESIZE notification messages
SetBackgroundColor()	Sets the control's background color
SetDefaultCharFormat()	Sets the default character format
SetEventMask()	Sets the control's event mask
SetModify()	Toggles the control's modification flag
SetOLECallback()	Sets the control's IRichEditOleCallback COM object
SetOptions()	Sets the control's options
SetParaFormat()	Sets the selection's paragraph format
SetReadOnly()	Disables editing in the control
SetRect()	Sets the control's formatting rectangle
SetSel()	Sets the selected text
SetSelectionCharFormat()	Sets the selected text's character format
SetTargetDevice()	Sets the control's target output device
SetWordCharFormat()	Sets the current word's character format
StreamIn()	Brings text in from an input stream
StreamOut()	Stores text in an output stream
Undo()	Undoes the last edit

---



[Brief](#)   [Full](#)  
+ [Advanced Search](#)  
+ [Search Tips](#)



[an error occurred while processing this directive]

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Listing 13.7 LST13\_07.TXT—Changing Paragraph Formats

```

void CRicheditView::OnLeft()
{
    PARAFORMAT paraFormat;
    paraFormat.cbSize = sizeof(PARAFORMAT);
    paraFormat.dwMask = PFM_ALIGNMENT;
    paraFormat.wAlignment = PFA_LEFT;
    m_richEdit.SetParaFormat(paraFormat);
}

void CRicheditView::OnCenter()
{
    PARAFORMAT paraFormat;
    paraFormat.cbSize = sizeof(PARAFORMAT);
    paraFormat.dwMask = PFM_ALIGNMENT;
    paraFormat.wAlignment = PFA_CENTER;
    m_richEdit.SetParaFormat(paraFormat);
}

void CRicheditView::OnRight()
{
    PARAFORMAT paraFormat;
    paraFormat.cbSize = sizeof(PARAFORMAT);
    paraFormat.dwMask = PFM_ALIGNMENT;
    paraFormat.wAlignment = PFA_RIGHT;
    m_richEdit.SetParaFormat(paraFormat);
}
    
```

## Using the MFC *CRichEditView* Class

As I mentioned previously, there's an easier way to add a rich edit control to your application: Derive your view class from MFC's *CRichEditView* class. When you do this with AppWizard, you get an almost complete application, one that not only enables the user to type and edit text but also incorporates sophisticated cut-and-paste features, as well as file saving and loading and even printing and print preview. To see all of this work, first create the *MFCRichEdit* application by performing the following steps.

Brief Full

- Advanced
- Search
- Search Tips

 **BROWSE**  
BY TOPIC



The complete source code and executable file for the MFCRichEdit application can be found in the CHAP13\MFCRichEdit directory on this book's CD-ROM.

1. Start a new AppWizard project workspace called MFCRichEdit, as shown in Figure 13.10.



**FIG. 13.10** Here's how to use AppWizard to start a project workspace called MFCRichEdit.

2. Give the new project the following settings in the AppWizard dialog boxes:
  - Step 1: Single document
  - Step 2: Default settings
  - Step 3: Default settings
  - Step 4: Default settings
  - Step 5: Default settings
  - Step 6: Change the base class for CMFCRichEditView to CRichEditView(see Figure 13.11).



**FIG. 13.11** You want to derive your view class from MFC's CRichEditView.

3. Add a button to the application's toolbar, as shown in Figure 13.12. Give the button the ID ID\_UNDERLINED.



**FIG. 13.12** Use the drawing tools to add a button to the application's toolbar.

4. Use ClassWizard to associate the ID\_UNDERLINED command with the OnUnderlined() message response function, as shown in Figure 13.13. Make sure that you have CMFCRichEditView selected in the Class Name box before you add the function.



**FIG. 13.13** Add the OnUnderlined() message response function to the view class.

5. Click the Edit Code button, and then add the lines shown in Listing 13.8 to the new OnUnderlined() function, right after the TODO: Add your command handler code here comment:

**Listing 13.8 LST13\_08.CPP—Code for the OnUnderlined() Function**

---

```
CHARFORMAT charFormat = GetCharFormatSelection();

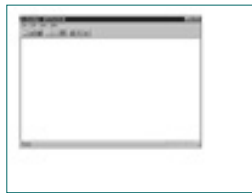
if (charFormat.dwEffects & CFM_UNDERLINE)
    charFormat.dwEffects = 0;
else
    charFormat.dwEffects = CFE_UNDERLINE;

SetCharFormat(charFormat);
```

---

You've now completed the MFCRichEdit application. To compile the application, choose Developer Studio's Build, Build command. You can then run the program by choosing the Build, Execute command. When you do, the application's main window appears, as shown in Figure 13.14. Take some time to experiment with the application. As you'll discover, the application automatically takes care of editing, file-saving, file-loading, print, and print-preview features.

Although MFC's CRichEditView class automatically takes care of most text-handling features, you still have to provide your own code for doing things like setting character and paragraph styles. These tasks, however, are accomplished similarly to the way in which they were handled in the original program, RichEdit. The difference is that, rather than calling a rich edit control's member functions directly, you call the CRichEditView object's member functions. The sample program demonstrates only text underlining, but that should be enough to show how you can easily add other features.



**FIG. 13.14** MFCRichEdit application provides a host of features with little effort from the programmer.

---

**Note:**

An important advantage of using the CRichEditDoc and CRichEditView classes is that CRichEditDoc and CRichEditView manage the document and view using MFC's document/view architecture, enabling you to save your documents in RTF format and to reload them with the formatting intact.

---

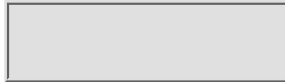
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 14

# The Animation Control

- Learn to create an animation control in an application's main window
- Discover how to program an animation control
- Find out how to place an animation control in a dialog box

Windows 95 is loaded with animation sequences. These sequences include the circling magnifying glass that you see when Windows is searching for a file, as well as the flying files that move from folder to folder when you copy files. These animation sequences have one thing in common: They are produced using Windows' new animation control. You can use the animation control in your own programs to provide dynamic feedback to the user or to just do something a little different with a dialog box or window. As you'll learn in this chapter, the animation control is surprisingly easy to use.

## Introducing the Animation Control

As I mentioned previously, the animation control is used throughout Windows 95. Figure 14.1 shows the circling hourglass, and Figure 14.2 shows the flying files. When you create your own animation sequences, they must be stored as a special AVI audio/video file. These files must contain only a single video stream with at least one animation frame and must be either uncompressed or compressed using RLE-8 compression.

---

### Note:

The AVI file can contain an audio stream. However, the animation control ignores all audio information.

---

Finding animation sequences that work with the animation control can be tough. All of Windows' animation sequences are hidden away somewhere (that is, they aren't available as AVI files). Moreover, most of the animation sequences you can find in the AVI format do not meet the animation control's strict requirements. Most often, you have to create your own AVI files. You can do this with special video software that comes with the Video for Windows SDK. Other, third-party, video software packages might also be able to create these special AVI files.





**FIG. 14.1** The circling magnifying glass indicates that Windows is searching for a file.



**FIG. 14.2** The flying files appear when Windows copies files.

Luckily, the Visual C++ CD-ROM includes a few suitable AVI files along with its sample applications. In this chapter, you'll use one of those animation sequences as you learn to program the animation control, which, under MFC, is represented by the `CAnimateCtrl` class.

## Creating the Animate Application

To get started quickly, you'll create the first of this chapter's sample programs using Developer Studio's AppWizard. This application, called *Animate*, displays an animation sequence in an animation control and provides toolbar buttons for manipulating the animation. Perform the following steps to create the *Animate* application.



The complete source code and executable file for the *Animate* application can be found in the `CHAP14\Animate` directory on this book's CD-ROM.

1. Start a new AppWizard project workspace called *Animate*, as shown in Figure 14.3.
2. Give the new project the following settings in the AppWizard dialog boxes. When you're finished, the New Project Information dialog box should look like Figure 14.4.

Step 1: Single document

Step 2: Default settings

Step 3: Default settings

Step 4: Turn off Printing and Print Preview

Step 5: Default settings

Step 6: Default settings



**FIG. 14.3** Start an AppWizard project workspace called *Animate*.





**FIG. 14.8** Add message response functions for the other toolbar buttons.

9. Use ClassWizard to associate the WM\_CREATE Windows message with the OnCreate() message response function in the view class, as shown in Figure 14.9.
10. Click the Edit Code button, and then add the following lines to the new OnCreate() function, right after the TODO: Add your specialized creation code here comment:

```
m_seekPos = 0;
m_animateCtrl.Create(WS_CHILD | WS_VISIBLE,
    CRect(10, 10, 100, 100), this, 111);
```



**FIG. 14.9** Add the OnCreate() function to the view class.

11. Find the OnOpenanimation() function in the AnimateView.cpp file, and then add the following line, right after the TODO: Add your command handler code here comment:

```
m_animateCtrl.Open("filecopy.avi");
```

12. Find the OnStartanimation() function, and then add the following line, right after the TODO: Add your command handler code here comment:

```
m_animateCtrl.Play((UINT)0, (UINT)-1, (UINT)-1);
```

13. Find the OnStopanimation() function, and then add the following line, right after the TODO: Add your command handler code here comment:

```
m_animateCtrl.Stop();
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

Table 14.1 Animation Control Styles

Style	Description
ACS_AUTOPLAY	Plays the clip automatically when it’s opened.
ACS_CENTER	Centers the animation frames inside the control without resizing or repositioning the control. Without this flag, the animation control resizes itself to accommodate the animation frames.
ACS_TRANSPARENT	Draws the animation sequence with a transparent background.

The flags in Table 14.1 are mostly self-explanatory. The ACS\_TRANSPARENT flag is helpful when you’re displaying an animation sequence such as FILECOPYY.AVI in a dialog box. (See the section “Adding an Animation Control to a Dialog Box,” later in this chapter.) When you add this flag, the background color used in each animation frame is replaced by the background color defined by Windows (usually gray), which makes the background blend in with the dialog box’s background. You add the ACS\_TRANSPARENT (or any other) flag by ORing it in with the other flags, of which WS\_CHILD and WS\_VISIBLE are usually required, like this:

```
m_animateCtrl.Create(WS_CHILD | WS_VISIBLE | ACS_TRANSPARENT,
    CRect(10, 10, 100, 100), this, 111);
```

Manipulating the Animation Control

As soon as the animation control is created, you might want to initialize related variables in some way. For example, the Animate application uses the member variable m\_seekPos to help implement the single-stepping functions. That variable is initialized to 0 in OnCreate(). The CAnimateCtrl class features a number of member functions that enable you to manipulate the control in limited ways. Those member functions and their descriptions are listed in Table 14.2.

Table 14.2 Member Functions of the CAnimateCtrl Class

Function	Description
Close()	Closes the animation sequence.
Create()	Creates the control.
Open()	Loads an animation sequence.
Play()	Plays the animation sequence.
Seek()	Displays a given animation frame.
Stop()	Stops playing the animation sequence.

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

**14.** Find the OnStepforward() function, and then add the lines shown in Listing 14.1, right after the TODO: Add your command handler code here comment.

#### **Listing 14.1 LST14\_01.CPP—Code for the OnStepforward() Function**

```
++m_seekPos;
if (m_seekPos > 16)
    m_seekPos = 0;
m_animateCtrl.Seek(m_seekPos);
```

**15.** Find the OnStepbackward() function, and then add the lines shown in Listing 14.2, right after the TODO: Add your command handler code here comment:

#### **Listing 14.2 LST14\_02.CPP—Code for the OnStepbackward() Function**

```
--m_seekPos;
if (m_seekPos < 0)
    m_seekPos = 16;
m_animateCtrl.Seek(m_seekPos);
```

**16.** Load the AnimateView.h file, and then add the following lines to the class's Attributes section, right after the line CAnimateDoc\* GetDocument():

```
protected:
    CAnimateCtrl m_animateCtrl;
    int m_seekPos;
```

**17.** Copy the FILECOPYY.AVI file from your Visual C++ CD's DevStudio\Vc\Samples\ mfc\general\cmnctrls directory to your Animate project directory.

You've now completed the Animate application. To compile the application, select Developer Studio's **B**uild, **B**uild command. You then can run the program by selecting the **B**uild, **E**xecute command. When you do, the application's main window appears (see Figure 14.10). Select the toolbar's Open button to load the animation sequence. Then select the Start button to get the animation going. When you do, files start flying between the two folders displayed on the screen (see Figure 14.11). You can stop the animation at any time by clicking the toolbar's Stop button. The arrow buttons enable you to

single-step through the animation sequence, either forward or backward.



**FIG. 14.10** This is Animate when you first run it.



**FIG. 14.11** This is Animate after you open and run the animation sequence.

## Exploring the Animate Application

As you created the Animate application, you probably were able to figure out how most of the code worked. An animation control, after all, is limited in what it can do, which makes it easy to program. There might be a few of the details, however, that you're not clear on, so in the following sections, you can examine the Animate application more closely.

### Creating the Animation Control

In the Animate application, the animation control is created in the `OnCreate()` member function, which MFC calls when Windows sends the `WM_CREATE` message. In `OnCreate()`, the program creates the control by calling its `Create()` member function, like this:

```
m_animateCtrl.Create(WS_CHILD | WS_VISIBLE,  
    CRect(10, 10, 100, 100), this, 111);
```

This function's four arguments are the control's style flags, the control's position and size, a pointer to the control's parent window, and the control's ID. The style constants include the same constants that you would use for creating any type of window, with the addition of special styles used with animation controls. Table 14.1 lists these special styles.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)[ITKNOWLEDGE](#)

Brief Full

- Advanced
- Search
- Search Tips

[BROWSE](#)[BY TOPIC](#)

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

As you can tell from Table 14.2, there's not a heck of a lot you can do with an animation control. However, the Animate program shows you how to use the control's various member functions (all except Close(), that is). You already know how to create the control. To load an animation sequence, you call the CAnimateCtrl class's Open() member function, like this:

```
BOOL result = m_animateCtrl.Open("filecopy.avi");
```

This function requires the animation sequence's file name or resource ID as its single argument. If the animation sequence loads successfully, Open() returns TRUE; otherwise, it returns FALSE.

When the animation sequence has been opened successfully, you can play it by calling the Play() member function:

```
BOOL result = m_animateCtrl.Play((UINT)0, (UINT)-1, (UINT)-1);
```

This function's three arguments are the index of the frame (starting at 0) at which the animation sequence should start playing, the number of the frame at which playing should end (with -1 indicating that the animation should end with the last frame), and the number of times to play the animation sequence. A value of -1 for this last argument plays the sequence continuously. The Play() method returns TRUE if everything went okay and FALSE if there were an error.

**Note:**

Note that all three of Play()'s arguments are unsigned integers. How can -1 be an unsigned integer? Don't ask me, ask Microsoft. You'll get compiler warnings, however, if you don't make the correct type casts.

You can stop playing an animation sequence by calling the Stop() member function, like this:

```
BOOL result = m_animateCtrl.Stop();
```

This function requires no arguments and returns TRUE if successful and FALSE if unsuccessful.

One interesting thing you can do with the Animate application is to advance or reverse frame-by-frame through the animation sequence. This is done by calling the CAnimateCtrl object's Seek() member function, which enables you to display any frame of animation that you like. The Animate application uses Seek() in conjunction with a member variable, called m\_seekPos, that keeps track of the currently selected frame. The program first increments m\_seekPos:

```
++m_seekPos;
```

The program then checks that the variable is still within the correct range for the animation sequence:

```
if (m_seekPos > 16)
    m_seekPos = 0;
```

As soon as `m_seekPos` has a valid value, a call to `Seek()` displays the selected frame of animation, like this:

```
m_animateCtrl.Seek(m_seekPos);
```

The `Seek()` member function's single argument is the zero-based index of the frame to display. A value of `-1` indicates that the last frame should be displayed. The index must be less than 65,535.

## Adding an Animation Control to a Dialog Box

Chances are, when you need the animation control, it'll be in some sort of dialog box. When you're using Visual C++'s dialog box editor, adding an animation control to a dialog box is quick and easy. To create an application that displays a dialog box with an animation control, follow these steps.



The complete source code and executable file for the `Animate2` application can be found in the `CHAP14\Animate2` directory on this book's CD-ROM.

1. Start a new AppWizard project workspace called `Animate2`, as shown in Figure 14.12.



**FIG. 14.12** Start an AppWizard project workspace called `Animate2`.

2. Give the new project the following settings in the AppWizard dialog boxes. When you're finished, the New Project Information dialog box should look like Figure 14.13.

- Step 1: Single document
- Step 2: Default settings
- Step 3: Default settings
- Step 4: Turn off everything except 3D Controls
- Step 5: Default settings
- Step 6: Default settings



**FIG. 14.13** These are the AppWizard settings for the `Animate2` project.



3. Using the resource editor, remove the application's Edit menu. Also remove all items from the File menu except for Exit.
4. Add a Test menu to the program, with a single item called Dialog Box (see Figure 14.14). Set the Dialog Box command's ID to ID\_TEST\_DIALOGBOX.
5. Delete all accelerators from the application's resources.
6. Create the dialog box shown in Figure 14.15. The empty square in the upper left of the dialog box is an animation control, which you can create from the toolbox just as you create any kind of control. (On the toolbox, the animation control looks like a film strip.)
7. Select the animation control and press Enter. Set the Animate button's ID to IDC\_ANIMATE1 and then turn on the control's Transparent style (see Figure 14.16).
8. Select the dialog box, and press Ctrl+W to bring up ClassWizard. Create a class called CAnimateDlg for the dialog box, as shown in Figure 14.17.



**FIG. 14.14** Use the resource editor to edit the application's menus.



**FIG. 14.15** This is the dialog box that displays the animation.



**FIG. 14.16** You need to set the animation control's style such that it doesn't appear in the dialog box with a colored background.



**FIG. 14.17** To control the dialog box from your program, you need to create a class for the dialog box.

9. Use ClassWizard to associate the ID\_TEST\_DIALOGBOX command with the OnTestDialogbox() message response function, as shown in Figure 14.18. Make sure that you have CAnimate2View selected in the Class Name box before you add the function.



**FIG. 14.18** Add the OnTestDialogbox() message response function to the view class.

**10.** Click the Edit Code button, and then add the following lines to the new OnTestDialogbox() function, right after the TODO: Add your command handler code here comment:

```
CAnimatedDlg dlg;
dlg.DoModal();
```

**11.** Near the top of the CAnimate2View file, add the following line, right after the #endif line:

```
#include "animatedlg.h"
```

**12.** Use ClassWizard to associate the dialog box's IDC\_ANIMATE button with the OnAnimate() message response function, as shown in Figure 14.19. Make sure you have the CAnimatedDlg class selected in the Class Name box.



**FIG. 14.19** Add the OnAnimate() message response function.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

**13.** Click the Edit Code button, and then add the following lines to the new OnAnimate() function, right after the TODO: Add your control notification handler code here comment:

```

CAnimateCtrl* pCtrl = (CAnimateCtrl*)GetDlgItem( IDC_ANIMATE1 );
pCtrl->Open( "filecopy.avi" );
pCtrl->Play( (UINT)0, (UINT)-1, (UINT)-1 );
  
```

**14.** Copy the FILECOPY.AVI file from your Animate project directory to your Animate2 project directory.

You've now completed the Animate2 application. To compile the application, select Developer Studio's Build, Build command. You then can run the program by selecting the Build, Execute command. When you do, the application's main window appears. Select the Test, Dialog Box command to display the dialog box containing the animation control. Then select the Animate button. When you do, the dialog box loads and runs the animation sequence (see Figure 14.20). You can stop the animation at any time by clicking the Cancel button.



**FIG. 14.20** The animation control in the dialog box now displays the flying files.

## Exploring the Animate2 Application

There's not much to know about the Animate2 application, as far as its animation control goes. Still, a quick rundown of the differences between this application and its predecessor is in order.

First, you don't have to call the animation control's Create() function when the control is part of a dialog box. MFC takes care of that detail for you. However, you do still need to tell the dialog box what to do with the animation control. Animate2 does this in response to the Animate button, which causes MFC to call the CAnimateDlg class's OnAnimate() member function. In that function, the program first associates an object of the CAnimateCtrl class with the dialog box's animation control, like this:

```

CAnimateCtrl* pCtrl = (CAnimateCtrl*)GetDlgItem( IDC_ANIMATE1 );
  
```

The program can then manipulate the control, which it does by calling the control's Open() and Play() member functions through the pointer returned by the previous call to GetDlgItem():

```

pCtrl->Open( "filecopy.avi" );
pCtrl->Play( (UINT)0, (UINT)-1, (UINT)-1 );
  
```

---

**Note:**

Notice the difference that setting the animation control's Transparent style makes in this program. The animation control's background matches that of the dialog box, rather than being some other color.

---

That's all there is to it! If you want the animation to run when the dialog box first appears, just associate the WM\_INITDIALOG Windows message with the OnInitDialog() member function, as shown in Figure

14.21. Place the animation control member-function calls in OnInitDialog(), as shown in Listing 14.3. Make sure to call the base class's version of OnInitDialog().



**FIG. 14.21** Use OnInitDialog() to get an animation going while MFC is initializing the dialog box.

---

**Listing 14.3 LST14\_03.CPP—The OnInitDialog() Message Response Function**

---

```
BOOL CAnimatedDlg::OnInitDialog()  
{  
    CDialog::OnInitDialog();  
  
    // TODO: Add extra initialization here  
  
    CAnimateCtrl* pCtrl = (CAnimateCtrl*)GetDlgItem(IDC_ANIMATE1);  
    pCtrl->Open("filecopy.avi");  
    pCtrl->Play((UINT)0, (UINT)-1, (UINT)-1);  
  
    return TRUE; // return TRUE unless you set the focus to a control  
                // EXCEPTION: OCX Property Pages should return FALSE  
}
```

---

One of the trickiest parts of using the animation control is finding or creating an AVI file that suits your needs. The process of creating AVI files is beyond the scope of this book, but you can find tools on the market that'll help you with this process. The first place to look might be the Microsoft Video for Windows SDK.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Part III

# Advanced Programming with MFC

## Chapter 15

# MFC File Handling

- Learn how persistent objects help you keep documents up-to-date
- Explore how a standard Document/View application deals with persistence
- Discover how to create your own persistent class
- Learn how to use MFC's CFile class to read and write files directly
- See how to create your own archive objects

One of the most important things a program must do is save a user's data after that data has been altered in some way. If the application does not have the capability to save edited data, the work the user performs with an application exists only as long as the application is running, vanishing the instant the user exits the application. Not a good way to get work done! In many cases, especially when using AppWizard to create an application, Visual C++ provides much of the code you need to save and load data. However, in some cases—most notably when you create your own object types—you have to do a little extra work to keep your user's files up-to-date.

## Objects and Persistence

When you're writing an application, you deal with many different types of objects. Some of your data objects might be simple types like integers and characters. Other objects might be instances of classes, like strings from the CString class or even objects created from your own custom classes. When using objects in applications that must create, save, and load documents, you need a way to save and load the state of those objects so that you can re-create them exactly as the user left them at the end of the last session.

An object's capability to save and load its state is called ***persistence***. Almost all of the MFC classes are persistent because they are derived either directly or indirectly from MFC's CObject class, which provides the basic functionality for saving and loading an object's state. You've already had some experience with this feature of Visual C++'s MFC. In the following section, though, you review

how MFC makes a document object persistent.

## The File Demo Application

When you create a program using Visual C++'s AppWizard, you get an application that uses document and view classes to organize, edit, and display its data. As you know, the document object, which is derived from the CDocument class, is responsible for holding the application's data during a session and for saving and loading the data so that the document persists from one session to another.



In the CHAP15\FILE folder on this book's CD-ROM, you'll find the File Demo application, which demonstrates the basic techniques behind saving and loading data of an object derived from CDocument. When you run the application, you see the window shown in Figure 15.1. This window displays the contents of the current document. In this case, a document is a single string containing a short message.



**FIG. 15.1** The File Demo application demonstrates basic document persistence.

When the program first begins, the message is automatically set to the string Default Message. However, you can change this message to anything you like. To do this, select the Edit, Change Message command. You then see the dialog box shown in Figure 15.2. Type a new message in the edit box and click the OK button. The new message appears in the window.



**FIG. 15.2** You can use the Change Message dialog box to edit the application's message string.

If you choose to exit the program, the document's current state is lost. The next time you run the program, you have to change the message string again. To avoid this complication, you can save the document before exiting the program. Choose the File, Save command to do this (see Figure 15.3). After saving the document, you can reload it at any time by choosing File, Open.



**FIG. 15.3** Use the File menu to save and load documents.

## A Review of Document Classes

What you've just experienced is object persistence from the user's point of view. The programmer, of course, needs to know much more about how this persistence stuff works. Although you had some experience with document classes, you'll now review the basic concepts with an eye toward extending those concepts to your own custom classes.

- **See** "Understanding the Document Class" **p. 39**

When working with an application created by AppWizard, there are several steps you must complete to enable your document to save and load its state. Those steps, as they apply to an SDI (Single Document Interface) application, are as follows:

1. Define the data members that will hold the document's data.
2. Initialize the data members in the document class's OnNewDocument() member function.
3. Display the current document in the view class's OnDraw() member function.
4. Provide member functions in the view class that enable the user to edit the document.
5. Add, to the document class's Serialize() member function, the code needed to save and load the data that comprises the document.

## A Quick Look at File Demo's Source Code

In the File Demo application, the document class declares its document storage in its header file (FILEDOC.H) like this:

```
// Attributes
public:
    CString m_message;
```

In this case, the document's storage is nothing more than a single string object. Usually, your document's storage needs are much more complex. This single string, however, is enough to demonstrate the basics of a persistent document.

The document class must also initialize the document's data, which it does in the OnNewDocument() member function, as shown in Listing 15.1.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

### Listing 15.1 LST15\_01.CPP—Initializing the Document's Data

```

BOOL CFileDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    m_message = "Default Message";

    return TRUE;
}

```

With the document class's `m_message` data member initialized, the application can display the data in the View window, which it does in the view class's `OnDraw()` function, as shown in Listing 15.2.

### Listing 15.2 LST15\_02.CPP—Displaying the Document's Data

```

void CFileView::OnDraw(CDC* pDC)
{
    CFileDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here

    pDC->TextOut(20, 20, pDoc->m_message);
}

```

As long as the user is satisfied with the contents of the document, the program doesn't need to do anything else. But, of course, an application that doesn't enable the user to edit the application's documents is mostly useless. The File Demo application displays a dialog box that the user can use to edit the contents of the document, as shown in Listing 15.3.

### Listing 15.3 LST15\_03.CPP—Changing the Document's Data



```

void CFileView::OnEditChangemessage()
{
    // TODO: Add your command handler code here

    CChngDlg dialog(this);
    CFileDoc* pDoc = GetDocument();
    dialog.m_message = pDoc->m_message;

    int result = dialog.DoModal();

    if (result == IDOK)
    {
        pDoc->m_message = dialog.m_message;
        pDoc->SetModifiedFlag();
        Invalidate();
    }
}

```

---

This function, which responds to the application's Edit, Change Message command, displays the dialog box and, if the user exits the dialog box by clicking the OK button, transfers the string from the dialog box to the document's data member. The call to the document class's `SetModifiedFlag()` function notifies the class that its contents have been changed.

After the user has changed the document's contents, the data must be saved before exiting the application (unless, that is, the user doesn't want to save the changes he made). The document class's `Serialize()` function, shown in Listing 15.4, handles the saving and loading of the document's data.

---

**Listing 15.4 LST15\_04.CPP—The Document Class's `Serialize()` Function**

---

```

void CFileDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here

        ar << m_message;
    }
    else
    {
        // TODO: add loading code here

        ar >> m_message;
        UpdateAllViews(NULL);
    }
}

```

---

Because the CString class (of which m\_message is an object) defines the >> and << operators for transferring strings to and from an archive, it's a simple task to save and load the document class's data. If the document's data contained simple data types like integers or characters, it would also be easy to save and load the data. However, what if you've created your own custom class for holding the elements of a document? How can you make an object of this class persistent? You'll find the answers to these questions in the following section.

## Creating a Persistent Class

Suppose that you now want to enhance the File Demo application so that it contains its data in a custom class called CMessages. This class holds three CString objects, each of which must be saved and loaded if the application is going to work correctly. You have a couple of options. The first option is to save and load each individual string, as shown in Listing 15.5.

### Listing 15.5 LST15\_05.CPP—One Way to Save the New Class's Strings

---

```
void CFileDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here

        ar << m_messages.m_message1;
        ar << m_messages.m_message2;
        ar << m_messages.m_message3;
    }
    else
    {
        // TODO: add loading code here

        ar >> m_messages.m_message1;
        ar >> m_messages.m_message2;
        ar >> m_messages.m_message3;
        UpdateAllViews(NULL);
    }
}
```

---

In the preceding example, m\_messages is an object of the CMessages class. The CMessages class has three data members that make up the document's data. These data members, which are objects of the CString class, are called m\_message1, m\_message2, and m\_message3.

Although the solution shown in Listing 15.5 is workable, it's not particularly elegant. It would be better to make the CMessages class capable of creating persistent objects by completing the following steps:

1. Derive the new class from CObject.
2. Place the DECLARE\_SERIAL() macro in the class's declaration.

3. Place the `IMPLEMENT_SERIAL()` macro in the class's implementation.
4. Override the `Serialize()` function in the class.
5. Provide an empty, default constructor for the class.

In the following section, you explore an application that creates persistent objects exactly as described in the preceding steps.

## The File Demo 2 Application



The next sample application, File Demo 2, demonstrates the steps you take to create a class from which you can create persistent objects. You'll find this application in the `CHAP15\FILE2` folder on this book's CD-ROM. When you run the application, you see the window shown in Figure 15.4. The program's window displays the three strings that make up the document's data. These three strings are contained in a custom class.



**FIG. 15.4** The three strings displayed in the window are data members of a custom class

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

You can edit any of the three strings by choosing the Edit, Change Messages command. When you do, the dialog box shown in Figure 15.5 appears. Type the new string or strings that you want to display in the window, and then click the OK button. The program displays the edited strings and stores the new string values in the data object.



**FIG. 15.5** Use the Change Messages dialog box to edit the application's data.

The application's File menu contains the commands you need to save or load the contents of a document. If you save the changes you make before exiting the application, you can reload the document when you restart the application. In this case, unlike the first version of the program, the document class is using a persistent object—an object that knows how to save and load its own state—as the document's data.

## Looking at the CMessages Class

Before you can understand how the document class manages to save and load its contents successfully, you have to understand how the CMessages class, of which the document class's m\_messages data member is an object, works. As you examine this class, you'll see how the aforementioned five steps for creating a persistent class have been implemented. Listing 15.6 shows the class's header file.

### Listing 15.6 LST15\_06.CPP—The Header File of the CMessages Class

```
// messages.h
class CMessages : public CObject
{
    DECLARE_SERIAL(CMessages)
    CMessages(){};

protected:
    CString m_message1;
    CString m_message2;
    CString m_message3;
```

```

public:
    void SetMessage(UINT msgNum, CString msg);
    CString GetMessage(UINT msgNum);
    void Serialize(CArchive& ar);
};

```

---

First, notice that the CMessages class is derived from MFC's CObject class. Also, notice the DECLARE\_SERIAL() macro near the top of the class's declaration. This macro's single argument is the name of the class that you're declaring. MFC uses this macro to create the additional function declarations needed to implement object persistence. Next, the class declares a default constructor that requires no arguments. This constructor is necessary because MFC needs to be able to create objects of the class when loading data from a disk.

After the default constructor comes the class's data members, which are three objects of the CString class. The public member functions are next. SetMessage(), whose arguments are the number of the string to set and the string's new value, enables a program to change a data member. GetMessage(), on the other hand, is the complementary function, enabling a program to retrieve the current value of any of the strings. Its single argument is the number of the string to retrieve.

Finally, the class overrides the Serialize() function, where all the data saving and loading takes place. The Serialize() function is the heart of a persistent object, with each persistent class implementing it in a different way. Listing 15.7 is the class's implementation file, which defines the various member functions.

---

**Listing 15.7 LST15\_07.CPP—The Implementation File of the CMessages Class**

---

```

// messages.cpp

#include "stdafx.h"
#include "messages.h"

IMPLEMENT_SERIAL(CMessages, CObject, 1)

void CMessages::SetMessage(UINT msgNum, CString msg)
{
    if (msgNum == 1)
        m_message1 = msg;
    else if (msgNum == 2)
        m_message2 = msg;
    else if (msgNum == 3)
        m_message3 = msg;
}

CString CMessages::GetMessage(UINT msgNum)
{
    if (msgNum == 1)

```

```

        return m_message1;
    else if (msgNum == 2)
        return m_message2;
    else if (msgNum == 3)
        return m_message3;
    else
        return "";
}

void CMessages::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);

    if (ar.IsStoring())
    {
        ar << m_message1 << m_message2 << m_message3;
    }
    else
    {
        ar >> m_message1 >> m_message2 >> m_message3;
    }
}

```

---

The `IMPLEMENT_SERIAL()` macro is the counterpart to the `DECLARE_SERIAL()` macro. `IMPLEMENT_SERIAL()` instructs MFC how to define the functions that give the class its persistent capabilities. The macro's three arguments are the name of the class, the name of the immediate base class, and a schema number, which is like a version number. In most cases, you'll use 1 for the schema number.

There's nothing tricky about the `SetMessage()` and `GetMessage()` functions, which perform their assigned tasks in a straightforward fashion. The `Serialize()` function, however, might suggest a couple of questions. First, note that the first line of the body of the function calls the base class's `Serialize()` function. This is a standard practice for many functions that override functions of a base class. In this case, the call to `CObject::Serialize()` doesn't do much, because the `CObject` class's `Serialize()` function is empty. Still, calling the base class's `Serialize()` function is a good habit to get into, because you might not always be working with classes derived directly from `CObject`.

After calling the base class's version of the function, `Serialize()` saves and loads its data in much the same way that a document object does. Because the data members that must be serialized are `CString` objects, the program can use the `>>` and `<<` operators to write the strings to the disk.

## Using the *CMessages* Class in the Program

Now that you know how the `CMessages` class works, you can examine how it's used in the File Demo 2 application's document class. As you look over the document class, you see that the class uses the same steps to handle its data as the original File Demo application. The main difference is that it's now dealing

with a custom class, rather than simple data types or classes defined by MFC.  
First, the object is declared in the document class's declaration, like this:

```
// Attributes  
public:  
    CMessages m_messages;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Next, the program initializes the data object in the document class's OnNewDocument() class, as seen in Listing 15.8.

### Listing 15.8 LST15\_08.CPP—Initializing the Data Object

```

BOOL CFile2Doc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    m_messages.SetMessage(1, "Default Message 1");
    m_messages.SetMessage(2, "Default Message 2");
    m_messages.SetMessage(3, "Default Message 3");

    return TRUE;
}

```

Because the document class cannot directly access the data object's data members, it must initialize each string by calling the CMessages class's SetMessage() member function. The view class must edit the data the same way, by calling the CMessages object's member functions, as shown in Listing 15.9. The view class's OnDraw() function also calls the GetMessage() member function to access the CMessages class's strings.

### Listing 15.9 LST15\_09.CPP—Editing the Data Strings

```

void CFile2View::OnEditChangemessages()
{
    // TODO: Add your command handler code here

    CFile2Doc* pDoc = GetDocument();

    CChngDlg dialog(this);
    dialog.m_message1 = pDoc->m_messages.GetMessage(1);
    dialog.m_message2 = pDoc->m_messages.GetMessage(2);
    dialog.m_message3 = pDoc->m_messages.GetMessage(3);
}

```



```

        int result = dialog.DoModal();

        if (result == IDOK)
        {
            pDoc->m_messages.SetMessage(1, dialog.m_message1);
            pDoc->m_messages.SetMessage(2, dialog.m_message2);
            pDoc->m_messages.SetMessage(3, dialog.m_message3);
            pDoc->SetModifiedFlag();
            Invalidate();
        }
    }
}

```

---

The real action, however, happens in the document class's `Serialize()` function, where the `m_messages` data object is serialized out to disk. This is accomplished by calling the data object's own `Serialize()` function inside the document's `Serialize()`, as shown in Listing 15.10.

#### **Listing 15.10 LST15\_10.CPP—Serializing the Data Object**

---

```

void CFile2Doc::Serialize(CArchive& ar)
{
    m_messages.Serialize(ar);

    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here

        UpdateAllViews(NULL);
    }
}

```

---

As you can see, after serializing the `m_messages` data object, there's not much left to do in the document class's `Serialize()` function, except call `UpdateAllViews()` if data is being loaded rather than saved. Notice that the call to `m_messages.Serialize()` passes the archive object as its single parameter.

## **Reading and Writing Files Directly**

Although using MFC's built-in serialization capabilities is a handy way to save and load data, sometimes you need more control over the file-handling process. For example, you might need to deal with your files non-sequentially, something the `Serialize()` function and its associated `CArchive` object can't handle. In this case, you can handle files almost exactly as you did in your DOS programs, by creating, reading, and writing files directly. Even when you need to dig down to this level of file

handling, though, MFC offers help. Specifically, you can use the CFile class to handle files directly.

## The File Demo 3 Application

This book's CD-ROM contains an example program that shows how the CFile class works. You'll find this program in the CHAP15\FILE3 folder. When you run the program, you see the window shown in Figure 15.6. By choosing the Edit, Change Message command, you can edit the string that's displayed in the window (see Figure 15.7).



**FIG. 15.6** The File Demo 3 application uses the CFile class for direct file handling.



**FIG. 15.7** Use the Change Message dialog box to edit the application's display string.

Finally, you can save and load the displayed text string by choosing the File, Save and File, Open commands, respectively (see Figure 15.8).



**FIG. 15.8** The File menu enables you to save and load the application's display string.

## The CFile Class

MFC's CFile class encapsulates all of the functions you need to handle any type of file. Whether you want to perform common sequential data saving and loading or you want to construct a random-access file, the CFile class gets you there. Using the CFile class is a lot like handling files the old-fashioned C-style way, except that the class hides some of the busywork details from you so that you can get the job done quickly and easily. For example, you can create a file for reading with only a single line of code. Table 15.1 shows the CFile class's member functions and their descriptions.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 15.1 Member Functions of the CFile Class

Function	Description
Abort()	Immediately closes the file with no regard for errors
Close()	Closes the file
Duplicate()	Creates a duplicate file object
Flush()	Flushes data from the stream
GetFileName()	Gets the file’s filename
GetFilePath()	Gets the file’s full path
GetFileTitle()	Gets the file’s title (the filename without the extension)
GetLength()	Gets the file’s length
GetPosition()	Gets the current position within the file
GetStatus()	Gets the file’s status
LockRange()	Locks a portion of the file
Open()	Opens the file
Read()	Reads data from the file
Remove()	Deletes a file
Rename()	Renames the file
Seek()	Sets the position within the file
SeekToBegin()	Sets the position to the beginning of the file
SeekToEnd()	Sets the position to the end of the file
SetFilePath()	Sets the file’s path
SetLength()	Sets the file’s length
SetStatus()	Sets the file’s status
UnlockRange()	Unlocks a portion of the file
Write()	Writes data to the file

As you can see from the table, the CFile class offers plenty of file–handling power. The File Demo 3 application demonstrates how to call a few of the CFile class’s member functions. However, most of the other functions are just as easy to use.

Exploring the File Demo 3 Application

When the File Demo 3 application starts up, the program sets its display string to “Default Message,” sets the file path to “None,” and sets the file’s length to 0. This is all accomplished in the view class’s constructor. (For the sake of simplicity, all of the file handling is done in the view class.) When the user selects the Edit, Change Message command, the program displays the Change Message dialog box, which happens in the view class’s OnEditChangemessage() member function, as shown in Listing 15.11.

Listing 15.11 LST15\_11.CPP—Changing the Display String

```

void CFile3View::OnEditChangemessage()
{
// TODO: Add your command handler code here

    CChngDlg dialog(this);
    dialog.m_message = m_message;

    int result = dialog.DoModal();

    if (result == IDOK)
    {
        m_message = dialog.m_message;
        Invalidate();
    }
}

```

---

In this function, the program displays the dialog box, and, if the user exits the dialog box by clicking the OK button, sets the view class's `m_message` data member to the string entered into the dialog box. A call to `Invalidate()` ensures that the new string is displayed in the window. The process of displaying dialog boxes and extracting data from them should be very familiar to you by now.

When the user chooses the File, Save command, MFC calls the view class's `OnFileSave()` member function, which is shown in Listing 15.12.

---

**Listing 15.12 LST15\_12.CPP—The Application's `OnFileSave()` Function**

---

```

void CFile3View::OnFileSave()
{
// TODO: Add your command handler code here

    // Create the file.
    CFile file("TESTFILE.TXT",
    CFile::modeCreate | CFile::modeWrite);

    // Write data to the file.
    int length = m_message.GetLength();
    file.Write((LPCTSTR)m_message, length);

    // Obtain information about the file.
    m_filePath = file.GetFilePath();
    m_fileLength = file.GetLength();

    // Close the file and repaint the window.
    file.Close();
    Invalidate();
}

```

---



[Brief](#)   [Full](#)  
+ [Advanced Search](#)  
+ [Search Tips](#)



[an error occurred while processing this directive]

Table 15.2 The File Mode Flags

Flag	Description
CFile::modeCreate	Create a new file or truncate an existing file to length 0
CFile::modeNoInherit	Disallow inheritance by a child process
CFile::modeNoTruncate	When creating the file, do not truncate the file if it already exists
CFile::modeRead	Enable read operations only
CFile::modeReadWrite	Enable both read and write operations
CFile::modeWrite	Enable write operations only
CFile::shareCompat	Enable other processes to open the file
CFile::shareDenyNone	Enable other processes read or write operations on the file
CFile::shareDenyRead	Disable read operations by other processes
CFile::shareDenyWrite	Disable write operations by other processes
CFile::shareExclusive	Deny all access to other processes
CFile::typeBinary	Set binary mode for the file
CFile::typeText	Set text mode for the file

After creating the file, OnFileSave() gets the length of the current message and writes it out to the file by calling the CFile object’s Write() member function. This function requires, as arguments, a pointer to the buffer containing the data to write and the number of bytes to write. Notice the LPCTSTR casting operator in the call to Write(). This operator is defined by the CString class and extracts the string from the class.

Finally, the program calls the CFile object’s GetFilePath() and GetLength() member functions to get the file’s complete path and length, after which a call to Close() closes the file and a call to the view class’s Invalidate() function causes the window to display the new information.

Reading from a file is not much different from writing to one, as you can see in Listing 15.13, which shows the view class’s OnFileOpen() member function.

Listing 15.13 LST15\_13.CPP—Reading from the File

```

void CFile3View::OnFileOpen()
{
    // TODO: Add your command handler code here

    // Open the file.
    CFile file("TESTFILE.TXT", CFile::modeRead);

    // Read data from the file.
  
```

```
        char s[81];
        int bytesRead = file.Read(s, 80);
        s[bytesRead] = 0;
        m_message = s;

        // Get information about the file.
        m_filePath = file.GetFilePath();
        m_fileLength = file.GetLength();

        // Close the file and repaint the window.
        file.Close();
        Invalidate();
    }
```

---

This time the file is opened using the `CFile::modeRead` flag, which opens the file for read-operations only, after which the program creates a character buffer and calls the file object's `read()` member function to read data into the buffer. The `read()` function's two arguments are the address of the buffer and the number of bytes to read. The function returns the number of bytes actually read, which, in this case, is almost always less than the 80 requested. By using the number of bytes read, the program can add a 0 to the end of the character data, thus creating a standard C-style string that can be used to set the `m_message` data member. As you can see, the `OnFileOpen()` function calls the file object's `GetFilePath()`, `GetLength()`, and `Close()` member functions exactly as `OnFileSave()` did.

## Creating Your Own *CArchive* Objects

Although you can handle files using `CFile` objects, you can go a step further and create your own `CArchive` object that you can use exactly as you use the `CArchive` object in the `Serialize()` function. To do this, you create a `CFile` object and pass that object to the `CArchive` constructor, like this:

```
CFile file("FILENAME.EXT", CFile::modeRead);
CArchive ar(&file, CArchive::store);
```

After creating the archive object, you can use it just like the archive objects that are created for you by MFC. When you're through with the archive object, you must close both the archive and the file, like this:

```
ar.Close();
file.Close();
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

In OnFileSave(), the program first creates the file, as well as sets the file's access mode, by calling the CFile class's constructor. The constructor takes as arguments the name of the file to create and the file access mode flags. You can use several flags at a time simply by ORing their values together, as you can see in the previous listing. These flags, which describe how to open the file and which specify the types of valid operations, are defined as part of the CFile class and are listed in Table 15.2 along with their descriptions.

[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief Full  
• [Advanced Search](#)  
• [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 16 Using Bitmaps

- Learn the difference between device-dependent and device-independent bitmaps
- Learn to create an in-memory bitmap that's compatible with a window's display
- Discover how to display a bitmap
- Find out how to keep your window's display updated with a bitmap

The term *bitmap* is frequently misunderstood by new Windows programmers. For most people, a bitmap is a picture that can be displayed on the screen. While this definition is generally true, there are actually two main types of bitmaps used with Windows applications. The first, called *device-independent bitmaps* (DIBs), are found in picture files with a .BMP file extension and are the type of bitmap with which most people are familiar. You'll learn about DIBs in the next chapter.

The second type of bitmap, called a *device-dependent bitmap* (DDB), resides only in the computer's memory and is usually not a picture per se but rather some sort of image that a Windows application needs to create its display. In this chapter, you learn about DDBs and how they're used to build an application's display.

### Introducing Device-Dependent Bitmaps

As I just said, DIBs are the picture files that most people think of as bitmaps. DIBs are device independent because their file includes the color information needed to reproduce the picture on other devices. DDBs, on the other hand, don't include color tables, so you'll never (well, maybe I shouldn't say *never*) find DDBs saved to files. Instead, these types of images usually are created directly in the computer's memory and disappear when the application that created them terminates.

Because of their nature, DDBs are more utilitarian than DIBs. That is, the Windows programmer uses DDBs as tools for creating an application's display rather than displaying DDBs simply as pictures. Think of a Windows paint application. Often, a paint application enables the user to copy and paste a portion of the display. The user might, for example, copy the image of a tree and paste it down in various places on the display to create a forest. The tree image is a bitmap in the computer's memory. It's unlikely that the tree image will ever be saved to disk. On the other hand, the entire forest picture probably will be saved to disk as a DIB.

Another popular way to use DDBs (hereafter called *bitmaps*) is to use them to represent an application's entire display area in memory. The application makes changes to its display by drawing on the bitmap in memory and then copying the bitmap to the application's window. Using a bitmap in this way, an application can quickly update its display whenever it needs to without having to redraw the data from scratch. In the following section, you'll create a Windows application that handles its display in exactly this way.

## Creating the Bitmap Application

As mentioned, one of the most common uses for bitmaps is to store an application's display. You'll now create the Bitmap application, which uses a bitmap to store its window's contents. Whenever the Window needs a full repainting, such as when being uncovered from beneath another window, the application copies the bitmap from memory to the application's window. To create the Bitmap application, perform the following steps:

---

**Note:** The complete source code and executable file for the Bitmap application can be found in the CHAP16\Bitmap directory of this book's CD-ROM.

---

1. Start a new AppWizard project workspace called Bitmap, as shown in Figure 16.1.



**FIG. 16.1** Start an AppWizard project workspace called Bitmap.

2. Give the new project the following settings in the AppWizard dialog boxes. When you're finished, the New Project Information dialog box should look like Figure 16.2.

- Step 1: Single document
- Step 2: Default settings
- Step 3: Default settings
- Step 4: Turn off all options except 3D Controls
- Step 5: Default settings
- Step 6: Default settings



**FIG. 16.2** These are the AppWizard settings for the Bitmap project.

3. Using the resource editor, remove the application's Edit menu. Also remove all items from the File menu except Exit, as shown in Figure 16.3.



**FIG. 16.3** Use the resource editor to edit the application's menus.

4. Delete all accelerators from the application's resources.
5. Use ClassWizard to associate the WM\_CREATE Windows message with the OnCreate() message response function in the view class, as shown in Figure 16.4.



**FIG. 16.4** Add the OnCreate() function to the view class.

6. Click the Edit Code button and then add the lines shown in Listing 16.1 to the new OnCreate() function, right after the TODO: Add your specialized creation code here comment.

#### **Listing 16.1 LST16\_01.CPP—Code for the OnCreate() Function**

---

```
// Create a bitmap compatible with the window's DC.
CClientDC clientDC(this);
m_pBitmap = new CBitmap;
m_pBitmap->CreateCompatibleBitmap(&clientDC, 800, 600);

// Create a memory DC in which to store the bitmap.
CDC memDC;
memDC.CreateCompatibleDC(&clientDC);
memDC.SelectObject(m_pBitmap);

// Fill the bitmap with white.
CBrush brush(RGB(255,255,255));
memDC.FillRect(CRect(0,0,799,599), &brush);

// Draw the grid on the screen and bitmap.
DrawGrid(&memDC);
```

---

7. Use ClassWizard to associate the WM\_LBUTTONDOWN Windows message with the OnLButtonDown() message response function in the view class, as shown in Figure 16.5.



**FIG. 16.5** Add the OnLButtonDown() function to the view class.

8. Click the Edit Code button and then add the lines shown in Listing 16.2 to the new OnLButtonDown() function, right after the TODO: Add your message handler code here and/or call default comment.

#### **Listing 16.2 LST16\_02.CPP—Code for the OnLButtonDown() Function**

---

```
// If the user clicked in the grid...
if (ClickInsideGrid(point))
{
    // Calculate the square's column and row.
    UINT col = ((point.x - OFFSET) / SQUARESIZE) *
        SQUARESIZE + OFFSET;
    UINT row = ((point.y - OFFSET) / SQUARESIZE) *
        SQUARESIZE + OFFSET;

    // Draw the new wall square on the screen.
    DrawSquare(col, row);
}
```

---

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

9. Add the functions shown in Listing 16.3 to the end of the BitmapView.cpp file.

#### Listing 16.3 LST16\_03.CPP—New Functions for the CBitmapView Class

```
void CBitmapView::DrawGrid(CDC *pDC)
{
    // Draw the grid's vertical lines.
    for (int x=0; x<=COLSIZE; ++x)
    {
        pDC->MoveTo(SQUARESIZE*x+OFFSET, OFFSET);
        pDC->LineTo(SQUARESIZE*x+OFFSET, SQUARESIZE*COLSIZE+OFFSET);
    }

    // Draw the grid's horizontal lines.

    for (int y=0; y<=ROWSIZE; ++y)
    {
        pDC->MoveTo(OFFSET, SQUARESIZE*y+OFFSET);
        pDC->LineTo(SQUARESIZE*ROWSIZE+OFFSET, SQUARESIZE*y+OFFSET);
    }
}

BOOL CBitmapView::ClickInsideGrid(CPoint point)
{
    // Calculate width and height of grid in pixels.
    int gridSize = SQUARESIZE * ROWSIZE;

    // If the user clicked within the grid, return TRUE.
    if ((point.x < gridSize + OFFSET) && (point.x > OFFSET) &&
        (point.y < gridSize + OFFSET) && (point.y > OFFSET))
        return TRUE;

    // Otherwise, return FALSE.
    return FALSE;
}

void CBitmapView::DrawSquare(UINT x, UINT y)
{
    // Create a DC for the window and a memory
    // DC compatible with the window DC.
    CClientDC clientDC(this);
    CDC memDC;
    memDC.CreateCompatibleDC(&clientDC);

    // Select the grid bitmap into the memory DC.
    memDC.SelectObject(m_pBitmap);

    // Create a brush for the wall from the color array.
    CBrush* pBrush = new CBrush(RGB(255,0,0));
```

```

// Select the brush into both DCs.
CBrush* oldBrush1 = memDC.SelectObject(pBrush);
CBrush* oldBrush2 = clientDC.SelectObject(pBrush);

// Draw the square in both DCs.
memDC.Rectangle(x, y,
    x + SQUARESIZE + 1, y + SQUARESIZE + 1);
clientDC.Rectangle(x, y,
    x + SQUARESIZE + 1, y + SQUARESIZE + 1);

// Restore DCs and delete the brush.
memDC.SelectObject(oldBrush1);
clientDC.SelectObject(oldBrush2);
delete pBrush;
}

```

---

**10.** Add the following line to the CBitmapView class's destructor:

```
delete m_pBitmap;
```

This line deletes the bitmap that is created in the OnCreate() function.

**11.** Add the lines shown in Listing 16.4 to the OnDraw() function, right after the TODO: Add draw code for native data here comment.

---

#### **Listing 16.4 LST16\_04.CPP—Code for the OnDraw() Function**

---

```

// Create a memory DC that's compatible
// with the window's DC.
CDC memDC;
memDC.CreateCompatibleDC(pDC);

// Display the bitmap in the window.
memDC.SelectObject(m_pBitmap);
pDC->BitBlt(0, 0, 800, 600, &memDC, 0, 0, SRCCOPY);

```

---

**12.** Load the BitmapView.h file and then add the following lines to the class's Attributes section, right after the line CBitmapDoc\* GetDocument().

```
protected:
    CBitmap* m_pBitmap;
```

**13.** Add the following lines to the class's Implementation section, right after the protected keyword.

```
void DrawGrid(CDC* pDC);
BOOL ClickInsideGrid(CPoint point);
void DrawSquare(UINT x, UINT y);
```

**14.** Add the lines shown in Listing 16.5 to the top of the BitmapView.h file, right before the class's declaration.

---

#### **Listing 16.5 LST16\_05.CPP—Constant Declarations for the CBitmapView Class**

---

```
const SQUARESIZE = 12;
const ROWSIZE = 32;
const COLSIZE = 32;
```

```
const OFFSET = 2;
```

---

These lines define constants for some frequently used values in the program. SQUARESIZE is the size in pixels of each square in the grid; ROWSIZE is the number of rows in the grid; COLSIZE is the number of columns in the grid; and OFFSET is the number of pixels from the top and left that the grid is positioned in the main window.

**15.** Load the MainFrm.cpp file and then add the following lines to the PreCreateWindow() function, right before the function's return line:

```
// Set the window's size.  
cs.cx = 402;  
cs.cy = 440;
```

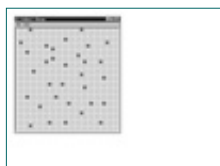
These lines set the width and height of the application's main window.

You've now completed the Bitmap application. To compile the application, select Developer Studio's **Build, Build** command. You can then run the program by selecting the **Build, Execute** command. When you do, the application's main window appears (see Figure 16.6).



**FIG. 16.6** The Bitmap application sports a grid of squares in its display.

Click anywhere in the on-screen grid, and the selected square turns red. Go ahead and paint some red squares (see Figure 16.7), and then minimize and restore the window. When the window restores, it instantly redraws its display. Thanks to the bitmap, the program doesn't need to redraw each individual square or even keep track of where those squares are located. (Of course, if you want to save your work to disk, you have to keep track of which squares are filled in.)



**FIG. 16.7** By clicking with the mouse, you can change the grid squares from white to red.

## Exploring the Bitmap Application

Now that you've had a chance to experiment with the Bitmap application, you probably want to take a look under the hood to see how this bitmap stuff really works. In the following sections, you'll examine each of the key functions in the Bitmap application. When you're through, you should have a solid understanding of how to use bitmaps in your own applications.

### Examining the OnCreate() Function

Whenever a new window is created, Windows sends the application a WM\_CREATE message. Because the window has a valid handle when WM\_CREATE is sent, the function OnCreate() (which responds to the WM\_CREATE message in an MFC program) is a good place to do initialization that requires a valid window. In the Bitmap application, the program uses OnCreate() to set up the bitmap that holds the window's display.

To update Bitmap's window as quickly as possible when it needs to be redrawn, the program stores a copy of the window's image in a bitmap in memory. For the program to display this bitmap in the window's client area, the bitmap must be selected into a device context that's compatible with the window's device context. So the first task in OnCreate() is to create a CClientDC object for the window:



```
CClientDC clientDC(this);
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

The CClientDC constructor requires a pointer to the window for which the DC is being created. In this case, you just use the this pointer. Then the program constructs a CBitmap object and uses it to create a bitmap that's compatible with the window's DC:

```
m_pBitmap = new CBitmap;
m_pBitmap->CreateCompatibleBitmap(&m_clientDC, 800, 600);
```

The CBitmap class's member function, CreateCompatibleBitmap(), takes as arguments the address of the device context with which the bitmap must be compatible and the width and height of the bitmap. CreateCompatibleBitmap() is an MFC version of a Windows API function of the same name.

Now, because this bitmap must be associated with a DC before you can draw on it, the program creates a memory DC in which to hold the bitmap:

```
CDC memDC;
memDC.CreateCompatibleDC(&m_clientDC);
```

The memory DC can be an object of the CDC class. Calling the class's CreateCompatibleDC() function (which is an MFC version of a Windows API function of the same name) ensures that the memory DC is compatible with the client window's DC. The function's single argument is the address of the DC with which the new DC should be compatible.

---

**Note:** Phrases like "create a compatible DC" sound very technical and can be confusing—which is unfortunate, because creating a compatible DC is really a simple process. As you know, a device context is simply a structure that describes the attributes for a device. These attributes include things such as pen color, a drawing surface of a specific size, a default font, and so on. Creating a compatible DC simply means creating a DC that has the same attributes as another DC.

---

Imagine that you're sitting at a table with a sheet of 8 1/2-x-11-inch typing paper, a red pencil, and a stencil for drawing Old English lettering. You can think of these drawing implements as a device context. Now suppose that a friend shows up who wants to help you with your project, so you give her a set of supplies just like yours. Giving your friend a second set of identical drawing supplies is similar to what happens when you create a compatible DC.

---

Next, because you want to draw on the bitmap that you created, you must associate the bitmap with the DC (called "selecting the bitmap into the DC"):

```
memDC.SelectObject(m_pBitmap);
```

The SelectObject() function, the CDC class's version of a Windows API function of the same name, takes care of this task. Its single argument is a pointer to the bitmap.

After selecting the bitmap into the DC, the program has an in-memory surface on which it can draw the image that will be transferred to the window's display. The first thing that the

program must do to prepare this surface is to paint it entirely white:

```
CBrush brush( RGB( 255, 255, 255 ) );  
memDC.FillRect( CRect( 0, 0, 799, 599 ), &brush );
```

Here, the program creates a white brush and uses the brush in a call to the FillRect() function. To construct a CBrush object, you need only supply the RGB values for the brush's colors. These values are the intensity of the red, green, and blue color components, respectively, which can be any value from 0 to 255. The higher the color value, the brighter the color component.

The FillRect() function, which is the CDC class's version of a Windows API function, takes two arguments: a CRect object containing the position and size of the rectangle to fill and the address of the brush with which to fill the rectangle.

At this point, the bitmap is a white rectangle. The next thing to do is to draw the lines that make up the grid. This line drawing is performed by a call to the CBitmapView class's DrawGrid() function:

```
DrawGrid( &memDC );
```

The DrawGrid() function contains two for loops. The first loop draws the vertical lines, as shown in Listing 16.6.

---

**Listing 16.6 LST16\_06.CPP—Drawing the Grid's Vertical Lines**

---

```
for (int x=0; x<=COLSIZE; ++x)  
{  
    pDC->MoveTo( SQUARESIZE*x+OFFSET, OFFSET );  
    pDC->LineTo( SQUARESIZE*x+OFFSET, SQUARESIZE*COLSIZE+OFFSET );  
}
```

---

As you can see, the line drawing is accomplished by calling the MoveTo() and LineTo() functions through the device-context pointer that was passed as one of DrawGrid()'s parameters. The MoveTo() function positions the drawing cursor at the X,Y coordinates given as the function's arguments. The LineTo() function then draws a line from the cursor position to the X,Y coordinates given as the LineTo() function's arguments.

As Listing 16.7 shows, the program draws the grid's horizontal lines in much the same way as it draws the vertical lines.

---

**Listing 16.7 LST16\_07.CPP—Drawing the Grid's Horizontal Lines**

---

```
for (int y=0; y<=ROWSIZE; ++y)  
{  
    pDC->MoveTo( OFFSET, SQUARESIZE*y+OFFSET );  
    pDC->LineTo( SQUARESIZE*ROWSIZE+OFFSET, SQUARESIZE*y+OFFSET );  
}
```

---

## Examining the OnDraw() Function

All of the bitmap finagling that you've done in OnCreate() only creates an image in memory. That image still doesn't appear in the Bitmap application's window. As you know, in an

AppWizard-generated application, it's the OnDraw() function that is charged with actually updating a window's display.

In the OnDraw() function, the program first creates a memory DC that's compatible with the DC passed to the function:

```
CDC memDC;  
memDC.CreateCompatibleDC(pDC);
```

The program then selects the bitmap (which contains the image to be drawn to the window) into the memory DC:

```
memDC.SelectObject(m_pBitmap);
```

Finally, a quick call to the window DC's BitBlt() function transfers the bitmap's image from the memory DC to the window's client area:

```
pDC->BitBlt(0, 0, 800, 600, &memDC, 0, 0, SRCCOPY);
```

BitBlt(), which is the CDC class's version of a Windows API function, takes eight arguments: the X,Y coordinates of the bitmap's destination rectangle; the width and height of the destination rectangle; a pointer to the source DC (the one containing the bitmap); the X,Y coordinates of the rectangle in the bitmap to copy; and a flag indicating how the source rectangle should be combined with the destination rectangle. The flag SRCCOPY means that the bitmap will completely overwrite any other image in the window.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Examining the *OnLButtonDown()* Function

As you just learned, the `OnDraw()` function takes care of repainting the application's display when the application receives a `WM_PAINT` message. However, what about when the user clicks a square? When this happens, the application must change the square's color from white to red. One way to accomplish this task is to draw the red square on the computer's in-memory bitmap and then transfer the bitmap to the display.

Another way is to draw the square directly on the screen, without bothering with `OnDraw()`, and also to update the bitmap in memory. In fact, every time you draw a red square in the Bitmap application's grid, that square gets drawn twice—once directly on the screen and once on the bitmap in memory. Whenever you want to place a new red square in the grid, you left-click the square. This causes the program's `OnLButtonDown()` function to be called.

`OnLButtonDown()` first checks whether the user's mouse click was inside the grid area:

```
if (ClickInsideGrid(point))
```

The `ClickInsideGrid()` function returns `TRUE` if the mouse pointer was over the grid and returns `FALSE` if the mouse pointer was over some other part of the window. If the user's mouse click wasn't over the grid, there's nothing to do, because you're not allowed to place a red square anywhere except within the boundaries of the grid.

If the mouse click is inside the grid, `OnLButtonDown()` calculates the square's coordinates, as shown in Listing 16.8.

### Listing 16.8 LST16\_08.CPP—Calculating the Square's Location in the Grid

```
UINT col = ((point.x - OFFSET) / SQUARESIZE) *
           SQUARESIZE + OFFSET;
UINT row = ((point.y - OFFSET) / SQUARESIZE) *
           SQUARESIZE + OFFSET;
```

The point data object (an instance of the `CPoint` class) is passed as a parameter to `OnLButtonDown()`. `point` holds the coordinates of the user's mouse click.

Finally, in `OnLButtonDown()`, a call to the `DrawSquare()` function draws a

rectangle in the grid square located at col,row:

```
DrawSquare(col, row);
```

## Examining the *DrawSquare()* Function

In the Bitmap application, it's the DrawSquare() function that actually places a red square on the grid. To avoid a sloppy screen update, however, DrawSquare() draws the square directly on the screen, as well as on the in-memory bitmap, giving the user an instant response to his or her mouse click.

First, DrawSquare() gets a device context for the view window, and then it creates a memory DC that's compatible with the window's DC:

```
CClientDC clientDC(this);  
CDC memDC;  
memDC.CreateCompatibleDC(&clientDC);
```

DrawSquare() then selects the bitmap that is holding the screen's image into the memory DC, where the program can draw on it:

```
memDC.SelectObject(m_pBitmap);
```

Next, the function needs a red brush with which to draw the square:

```
CBrush* pBrush = new CBrush(RGB(255,0,0));
```

A call to both of the DC's SelectObject() member functions associates the new brush with both DCs:

```
CBrush* oldBrush1 = memDC.SelectObject(pBrush);  
CBrush* oldBrush2 = clientDC.SelectObject(pBrush);
```

DrawSquare() can then call both DC objects' Rectangle() member functions to draw the rectangle on both the screen and on the in-memory bitmap, as shown in Listing 16.9.

### **Listing 16.9 LST16\_09.CPP—Drawing the Red Square on the Screen and on the Bitmap**

---

```
memDC.Rectangle(x, y,  
               x + SQUARESIZE + 1, y + SQUARESIZE + 1);  
clientDC.Rectangle(x, y,  
                  x + SQUARESIZE + 1, y + SQUARESIZE + 1);
```

---

Keeping the bitmap updated this way ensures that when OnDraw() needs to repaint the window's client area, the bitmap image will contain the correct image.

Finally, the function selects the old brushes back into the window and memory

DCs, freeing the brush used to draw on the bitmap. This brush is then deleted:

```
memDC.SelectObject(oldBrush1);  
clientDC.SelectObject(oldBrush2);  
delete pBrush;
```

Now that you know how to handle DDBs, you're ready to move forward into the much more complicated world of DIBs, which you do in the next chapter.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 17

# Manipulating Device-Independent Bitmaps

- Learn the format in which DIBs are stored in a disk file
- Create a custom class for handling DIBs
- Discover how to display a DIB
- Find out how to set Windows' color palette to a DIB's color table
- Learn about properly displaying bitmaps in a background application

If you've done any Windows programming, you know that bitmaps are everywhere. This is because the BMP graphics format is the only graphical-image format (not including icons, which are very limited) that Windows directly supports. Look through your Windows programming manuals, and you'll find functions such as `CreateBitmap()`, `LoadBitmap()`, `StretchDIBits()`, and `BitBlt()` that enable you to create, load, and display bitmaps on the screen, but you won't find similar functions for other graphics formats like PCX, TIF, and GIF.

The bitmaps you dealt with in the previous chapter were *device-dependent bitmaps* (DDBs). In this chapter, you'll learn about DDB's important cousin, *device-independent bitmaps* (DIBs).

## Reviewing DDBs and DIBs

Device-dependent bitmaps are graphical images that can be displayed on only one type of physical device. For example, when you use Windows functions such as `CreateBitmap()` and `LoadBitmap()`, you're creating in memory a bitmap image that is compatible with a certain device, usually the screen. These types of bitmaps are also sometimes called **GDI bitmaps** because Windows' GDI (graphics device interface) can handle them directly. DDBs are stored without color tables because they use the colors of their associated device. Moreover, DDBs usually reside only in memory, rather than as files on a disk.

Device-independent bitmaps are graphical images that can be displayed on many different devices. These types of bitmaps carry with them a color table that the current device must use to display the bitmap so that the bitmap looks similar from one device to another. For example, a DIB should look almost the same under Windows as it does under DOS or OS/2. Because DIBs are



generally portable between systems, you often find them as disk files. If you look in your Windows directory, for example, you'll see many files that have the BMP extension. These are DIBs. You can create your own DIBs using various types of paint programs, including Windows Paintbrush, which comes with every copy of Windows. You also can use Visual C++'s bitmap editor.

## The DIB Format

Whether a DIB is stored on disk or in memory, it has almost exactly the same structure. Actually, a DIB is made up of several different types of structures, one following the other. These structures include the BITMAPFILEHEADER, BITMAPINFO, BITMAPINFOHEADER, and RGBQUAD types. The following sections describe these structure types and how they're used in Windows programs.

### The *BITMAPFILEHEADER* Structure

At the beginning of a DIB file is the BITMAPFILEHEADER structure, which is defined by Windows as shown in Listing 17.1.

**Listing 17.1 LST17\_01.CPP—The BITMAPFILEHEADER Structure**

```
typedef struct tagBITMAPFILEHEADER {  
    WORD bfType;  
    DWORD bfSize;  
    WORD bfReserved1;  
    WORD bfReserved2;  
    DWORD bfOffBits;  
} BITMAPFILEHEADER;
```

Although this structure resides at the beginning of a DIB disk file, it need not be part of the DIB in memory. The first structure member, bfType, identifies the file as a DIB and should be the ASCII codes of the letters **BM**. In hex, the bfType word should be 4D42. Otherwise, the file is probably not a DIB. The second member, bfSize, is supposed to be the size of the DIB file in bytes. However, due to a mistake in the original Windows documentation, bfSize is not reliable and should be ignored. On the other hand, you can count on the bfOffBits member to contain the number of bytes from the start of the DIB file to the bitmap data. The BITMAPFILEHEADER structure is summarized in Table 17.1.

**Table 17.1 The BITMAPFILEHEADER Structure**

Member	Type	Description
bfType	WORD	Contains the ASCII values BM
bfSize	DWORD	The size of the file
bfReserved1	WORD	Always 0
bfReserved2	WORD	Always 0

bfOffBits	DWORD	The number of bytes from the beginning of the file to the bitmap
-----------	-------	------------------------------------------------------------------

---

## The *BITMAPINFO* Structure

Following the *BITMAPFILEHEADER* structure is the *BITMAPINFO* structure, which is defined by Windows as shown in Listing 17.2.

### Listing 17.2 LST17\_02.CPP—The *BITMAPINFO* Structure

---

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1];
} BITMAPINFO;
```

---

As you can see, this structure is made up of a header, represented by the *BITMAPINFOHEADER* structure and a color table, represented by an array of *RGBQUAD* structures.

## The *BITMAPINFOHEADER* Structure

A *BITMAPINFOHEADER* structure, also defined by Windows, looks like Listing 17.3.

### Listing 17.3 LST17\_03.CPP—The *BITMAPINFOHEADER* Structure

---

```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    DWORD biWidth;
    DWORD biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    DWORD biXPelsPerMeter;
    DWORD biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

---

The member *biSize* contains the size of the *BITMAPINFOHEADER* structure, which should be 40 bytes. The members *biWidth* and *biHeight* are the width and height of the bitmap in pixels. The member *biPlanes* is always set to 1. The *biBitCount* member, which indicates the number of bits per pixel, can be 1, 4, 8, or 24, which indicate monochrome, 16-color, 256-color, and 16.7 million color images. (In this book, you mostly use 256-color, or 8-bit, images.)

The `biCompression` member indicates the type of compression used with the bitmap image, where a 0 indicates no compression; a 1 indicates RLE-8 compression; and a 2 indicates RLE-4 compression. If you're not familiar with data compression techniques, don't worry about it. DIBs are rarely compressed. You'll usually find a 0 in the `biCompression` structure member.

The `biSizeImage` member is the size of the bitmap in bytes and is usually used only with compressed bitmaps. This value takes into account that the number of bytes in each row of a bitmap is always a multiple of 4. The rows are padded with blank bytes, when necessary, to ensure a multiple of 4. However, unless you're writing a program that creates DIBs, you don't need to deal with row padding and the code complications that arise from it.

The `biXPelsPerMeter` and `biYPelsPerMeter` members contain the horizontal and vertical number of pixels per meter of the intended display device, but are usually just set to 0. The `biClrUsed` and `biClrImportant` members, which contain the total number of colors used in the bitmap and the number of important colors in the bitmap, are also usually set to 0.

You might have noticed that `BITMAPINFOHEADER` structure members after `biBitCount` are likely to contain a 0, so after reading the structure from the disk file, you'll probably ignore the values stored in these structure members. In this chapter, you'll see how you can calculate any values you need—such as the number of colors used in the image—and store those values in the proper members for later retrieval. For easy reference, the `BITMAPINFOHEADER` structure is summarized in Table 17.2.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief
 Full

+ Advanced Search

+ Search Tips

[an error occurred while processing this directive]

**Table 17.2 The BITMAPINFOHEADER Structure**

Member	Type	Description
biSize	DWORD	Size in bytes of this structure
biWidth	DWORD	Bitmap's width in pixels
biHeight	DWORD	Bitmap's height in pixels
biPlanes	WORD	Always 1
biBitCount	WORD	Number of bits per pixel
biCompression	DWORD	Compression type: 0 = None, 1 = RLE-8, 2 = RLE-4
biSizeImage	DWORD	Bitmap's size in bytes
biXPelsPerMeter	DWORD	Horizontal pixels per meter
biYPelsPerMeter	DWORD	Vertical pixels per meter
biClrUsed	DWORD	Number of colors used
biClrImportant	DWORD	Number of important colors

**The *RGBQUAD* Structure**

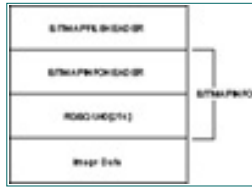
The final data structure, RGBQUAD, is defined by Windows as shown in Listing 17.4.

**Listing 17.4 LST17\_04.CPP—The RGBQUAD Structure**

```
typedef struct tagRGBQUAD {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
} RGBQUAD;
```

This structure simply contains the intensities of a color's red, green, and blue elements. Each color in a DIB is represented by an RGBQUAD structure. That is, a 16-color (4-bit) bitmap has a color table made up of 16 RGBQUAD structures, whereas a 256-color (8-bit) bitmap has a color table containing 256 RGBQUAD structures. The exception is 24-bit color images, which have no color table.

Following a DIB's BITMAPINFOHEADER structure is the bitmap's actual image data. The size of this data depends, of course, on the size of the image. Figure 17.1 illustrates the entire layout of a DIB file.



**FIG. 17.1** A DIB file contains several different structures.

## Introducing the CDib Class

Although MFC features classes for many graphical objects, you won't find one for DIBs. You don't, of course, necessarily need a special class to handle DIBs, but having such a class helps you organize your code into reusable modules. For that reason, in this chapter you develop a simple class called CDib, which reads DIBs from disk into memory and returns important information about the DIB.

### The CDib Class's Interface

The CDib class's interface is represented by its header file, which is shown in Listing 17.5.

#### Listing 17.5 CDIB.H—The CDib Class's Header File

```

////////////////////////////////////
// CDIB.H: Header file for the DIB class.
////////////////////////////////////

#ifndef __CDIB_H
#define __CDIB_H

class CDib : public CObject
{
protected:
    LPBITMAPFILEHEADER m_pBmFileHeader;
    LPBITMAPINFO m_pBmInfo;
    LPBITMAPINFOHEADER m_pBmInfoHeader;
    RGBQUAD* m_pRGBTable;
    BYTE* m_pDibBits;
    UINT m_numColors;

public:
    CDib(const char* fileName);
    ~CDib();

    DWORD GetDibSizeImage();
    UINT GetDibWidth();
    UINT GetDibHeight();

```

```

        UINT GetDibNumColors();
        LPBITMAPINFOHEADER GetDibInfoHeaderPtr();
        LPBITMAPINFO GetDibInfoPtr();
        LPRGBQUAD GetDibRGBTablePtr();
        BYTE* GetDibBitsPtr();

protected:
        void LoadBitmapFile(const char* fileName);

};

#endif

```

---

The CDib class's data members consist mostly of pointers to the various parts of a DIB, as shown in Listing 17.6.

---

**Listing 17.6 LST17\_06.CPP—Data Members of the CDib Class**

---

```

LPBITMAPFILEHEADER m_pBmFileHeader;
LPBITMAPINFO m_pBmInfo;
LPBITMAPINFOHEADER m_pBmInfoHeader;
RGBQUAD* m_pRGBTable;
BYTE* m_pDibBits;

```

---

The pointers in Listing 17.6 store the addresses of the DIB's BITMAPFILEHEADER, BITMAPINFO, and BITMAPINFOHEADER structures, as well as the addresses of the DIB's color table and image data. The final data member holds the number of colors in the DIB:

```

UINT m_numColors;

```

Like most classes, CDib has both a constructor and a destructor:

```

CDib(const char* fileName);
~CDib();

```

As you can see, you can create a CDib object by passing the file name of the bitmap that you want to load to the CDib constructor.

To enable you to obtain important information about a DIB after it is loaded, the CDib class features eight public member functions, as shown in Listing 17.7.

---

**Listing 17.7 LST17\_07.CPP—Member Functions of the CDib Class**

---

```

DWORD GetDibSizeImage();
UINT GetDibWidth();
UINT GetDibHeight();
UINT GetDibNumColors();
LPBITMAPINFOHEADER GetDibInfoHeaderPtr();

```

```
LPBITMAPINFO GetDibInfoPtr();  
LPRGBQUAD GetDibRGBTablePtr();  
BYTE* GetDibBitsPtr();
```

---

Table 17.3 lists each of the public member functions and what they do.

**Table 17.3 The CDib Class's Public Member Functions**

Name	Description
GetDibSizeImage()	Returns the size in bytes of the image
GetDibWidth()	Returns the DIB's width in pixels
GetDibHeight()	Returns the DIB's height in pixels
GetDibNumColors()	Returns the number of colors in the DIB
GetDibInfoHeaderPtr()	Returns a pointer to the DIB's BITMAPINFOHEADER structure
GetDibInfoPtr()	Returns a pointer to the DIB's BITMAPINFO structure
GetDibRGBTablePtr()	Returns a pointer to the DIB's color table
GetDibBitsPtr()	Returns a pointer to the DIB's image data

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 + [Advanced Search](#)  
 + [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Finally, the CDib class has a single protected member function that it calls internally to load a DIB file:

```
void LoadBitmapFile(const char* fileName);
```

You'll never call this member function directly.

### Programming the *CDib* Class

The code that defines the CDib class is found in the CDIB.CPP file, which is shown in Listing 17.8. In this file, each of the functions in the CDib class is defined.

#### Listing 17.8 CDIB.CPP—The Implementation of the CDib Class

```

////////////////////////////////////
// CDIB.CPP: Implementation file for the DIB class.
////////////////////////////////////

#include "stdafx.h"
#include "cdib.h"
#include "windowsx.h"

////////////////////////////////////
// CDib::CDib()
////////////////////////////////////
CDib::CDib(const char* fileName)
{
    // Load the bitmap and initialize
    // the class's data members.
    LoadBitmapFile(fileName);
}

////////////////////////////////////
// CDib::~~CDib()
////////////////////////////////////
CDib::~~CDib()
{
    // Free the memory assigned to the bitmap.
    GlobalFreePtr(m_pBmInfo);
}
  
```



```

////////////////////////////////////
// CDib::LoadBitmapFile()
//
// This function loads a DIB from disk into memory. It
// also initializes the various class data members.
////////////////////////////////////
void CDib::LoadBitmapFile
    (const char* fileName)
{
    // Construct and open a file object.
    CFile file(fileName, CFile::modeRead);

    // Read the bitmap's file header into memory.
    BITMAPFILEHEADER bmFileHeader;
    file.Read((void*)&bmFileHeader, sizeof(bmFileHeader));

    // Check whether the file is really a bitmap.
    if (bmFileHeader.bfType != 0x4d42)
    {
        AfxMessageBox("Not a bitmap file");
        m_pBmFileHeader = 0;
        m_pBmInfo = 0;
        m_pBmInfoHeader = 0;
        m_pRGBTable = 0;
        m_pDibBits = 0;
        m_numColors = 0;
    }
    // If the file checks out okay, continue loading.
    else
    {
        // Calculate the size of the DIB, which is the
        // file size minus the size of the file header.
        DWORD fileLength = file.GetLength();
        DWORD dibSize = fileLength - sizeof(bmFileHeader);

        // Allocate enough memory to fit the bitmap.
        BYTE* pDib =
            (BYTE*)GlobalAllocPtr(GMEM_MOVEABLE, dibSize);

        // Read the bitmap into memory and close the file.
        file.Read((void*)pDib, dibSize);
        file.Close();

        // Initialize pointers to the bitmap's BITMAPINFO
        // and BITMAPINFOHEADER structures.
        m_pBmInfo = (LPBITMAPINFO) pDib;
        m_pBmInfoHeader = (LPBITMAPINFOHEADER) pDib;

        // Calculate a pointer to the bitmap's color table.
        m_pRGBTable =
            (RGBQUAD*)(pDib + m_pBmInfoHeader->biSize);
    }
}

```

```

// Get the number of colors in the bitmap.
int m_numColors = GetDibNumColors();

// Calculate the bitmap image's size.
m_pBmInfoHeader->biSizeImage =
    GetDibSizeImage();

// Make sure the biClrUsed field
// is initialized properly.
if (m_pBmInfoHeader->biClrUsed == 0)
    m_pBmInfoHeader->biClrUsed = m_numColors;

// Calculate a pointer to the bitmap's actual data.
DWORD clrTableSize = m_numColors * sizeof(RGBQUAD);
m_pDibBits =
    pDib + m_pBmInfoHeader->biSize + clrTableSize;
}
}

////////////////////////////////////
// CDib::GetDibSizeImage()
//
// This function calculates and returns the size of the
// bitmap's image in bytes.
////////////////////////////////////
DWORD CDib::GetDibSizeImage()
{
    // If the bitmap's biSizeImage field contains
    // invalid information, calculate the correct size.
    if (m_pBmInfoHeader->biSizeImage == 0)
    {
        // Get the width in bytes of a single row.
        DWORD byteWidth = (DWORD) GetDibWidth();

        // Get the height of the bitmap.
        DWORD height = (DWORD) GetDibHeight();

        // Multiply the byte width by the number of rows.
        DWORD imageSize = byteWidth * height;

        return imageSize;
    }
    // Otherwise, just return the size stored in
    // the BITMAPINFOHEADER structure.
    else
        return m_pBmInfoHeader->biSizeImage;
}

////////////////////////////////////

```

```

// CDib::GetDibWidth()
//
// This function returns the width in bytes of a single
// row in the bitmap.
////////////////////////////////////
UINT CDib::GetDibWidth()
{
    return (UINT) m_pBmInfoHeader->biWidth;
}

////////////////////////////////////
// CDib::GetDibHeight()
//
// This function returns the bitmap's height in pixels.
////////////////////////////////////
UINT CDib::GetDibHeight()
{
    return (UINT) m_pBmInfoHeader->biHeight;
}

////////////////////////////////////
// CDib::GetDibNumColors()
//
// This function returns the number of colors in the
// bitmap.
////////////////////////////////////
UINT CDib::GetDibNumColors()
{
    if ((m_pBmInfoHeader->biClrUsed == 0) &&
        (m_pBmInfoHeader->biBitCount < 9))
        return (1 << m_pBmInfoHeader->biBitCount);
    else
        return (int) m_pBmInfoHeader->biClrUsed;
}

////////////////////////////////////
// CDib::GetDibInfoHeaderPtr()
//
// This function returns a pointer to the bitmap's
// BITMAPINFOHEADER structure.
////////////////////////////////////
LPBITMAPINFOHEADER CDib::GetDibInfoHeaderPtr()
{
    return m_pBmInfoHeader;
}

////////////////////////////////////
// CDib::GetDibInfoPtr()
//
// This function returns a pointer to the bitmap's
// BITMAPINFO structure.

```

```

////////////////////////////////////
LPBITMAPINFO CDib::GetDibInfoPtr()
{
    return m_pBmInfo;
}

////////////////////////////////////
// CDib::GetDibRGBTablePtr()
//
// This function returns a pointer to the bitmap's
// color table.
////////////////////////////////////
LPRGBQUAD CDib::GetDibRGBTablePtr()
{
    return m_pRGBTable;
}

////////////////////////////////////
// CDib::GetDibBitsPtr()
//
// This function returns a pointer to the bitmap's
// actual image data.
////////////////////////////////////
BYTE* CDib::GetDibBitsPtr()
{
    return m_pDibBits;
}

```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Loading a DIB into Memory

Look at the class's constructor, which is shown in Listing 17.9.

### Listing 17.9 LST17\_09.CPP—The CDib Class's Constructor

```
CDib::CDib(const char* fileName)
{
    // Load the bitmap and initialize
    // the class's data members.
    LoadBitmapFile(fileName);
}
```

You create a CDib object by calling the CDib class's constructor with the file name of the DIB that you want to load. The constructor passes this file name to the protected member function, LoadBitmapFile(), which actually loads the bitmap. LoadBitmapFile()'s code is fairly complex. In fact, it makes up the bulk of the CDib class's code. The other CDib member functions generally return values calculated by LoadBitmapFile().

First, LoadBitmapFile() constructs an MFC CFile object:

```
CFile file(fileName, CFile::modeRead);
```

This line of code not only constructs a CFile object named file but also opens the file (whose name is passed in the fileName argument) in the read-only mode.

Next, the function declares a BITMAPFILEHEADER structure and reads the DIB's file header into the structure:

```
BITMAPFILEHEADER bmFileHeader;
file.Read((void*)&bmFileHeader, sizeof(bmFileHeader));
```

The CFile object's Read() member function requires as its arguments a pointer to the buffer in which to store the data and the size of the buffer. In this case, the buffer is the BITMAPFILEHEADER structure.

After the preceding lines, the DIB's file header is stored in the bmFileHeader structure. The first task is to check whether the opened file is actually a bitmap. The function does this (as shown in Listing 17.10) by checking the bfType member for the value 0x4D42, which is the ASCII code for the letters **BM**.

### Listing 17.10 LST17\_10.CPP—Checking That the File Is a Bitmap

---

```
if (bmFileHeader.bfType != 0x4d42)
{
    AfxMessageBox("Not a bitmap file");
    m_pBmFileHeader = 0;
    m_pBmInfo = 0;
    m_pBmInfoHeader = 0;
    m_pRGBTable = 0;
    m_pDibBits = 0;
    m_numColors = 0;
}
```

---

If the `bfType` structure member does not contain the correct value, the function sets all the class's data members to 0 and exits. Otherwise, `LoadBitmapFile()` calls the `CFile` object's `GetLength()` member function to get the size of the opened file:

```
DWORD fileLength = file.GetLength();
```

The function then calculates the size of the DIB by subtracting the size of the file header from the size of the file:

```
DWORD dibSize = fileLength - sizeof(bmFileHeader);
```

The program uses the resulting value, `dibSize`, to allocate enough memory in which to store the DIB:

```
BYTE* pDib =
    (BYTE*)GlobalAllocPtr(GMEM_MOVEABLE, dibSize);
```

`GlobalAllocPtr()` is a Windows function that allocates memory and returns a pointer to that memory. It requires two arguments: a flag indicating how the memory should be allocated and the size of the memory block to be allocated. Please refer to your Windows programming manual for more information on `GlobalAllocPtr()`.

After allocating memory, the function calls the `CFile` object's `Read()` member function to read the DIB into memory and then closes the file by calling the `Close()` member function:

```
file.Read((void*)pDib, dibSize);
file.Close();
```

At this point, the DIB is stored in memory, and `LoadBitmapFile()` can now calculate the values that the class needs. The addresses of the DIB's `BITMAPINFO` and `BITMAPINFOHEADER` structures are identical, being the same address as the buffer in which the DIB is stored:

```
m_pBmInfo = (LPBITMAPINFO) pDib;
m_pBmInfoHeader = (LPBITMAPINFOHEADER) pDib;
```

The function then calculates a pointer to the color table by adding the size of the

BITMAPINFOHEADER structure to the address of the DIB in memory:

```
m_pRGBTable =  
    (RGBQUAD*)(pDib + m_pBmInfoHeader->biSize);
```

---

**Caution:**

When doing this type of pointer math, be careful that you're using the correct data types. For example, because `pDib` is a pointer to `BYTE`, the number of bytes stored in `biSize` is added to this pointer. However, if `pDib` were defined as a pointer to a `BITMAPINFO` structure, the value `biSize*sizeof(BITMAPINFO)` would be added to `pDib`. If you don't understand this, you should review how pointer math works.

---

Next, the function initializes the `m_numColors` data member, by calling the member function `GetDibNumColors()`:

```
int m_numColors = GetDibNumColors();
```

You'll see how `GetDibNumColors()` works a little later in this chapter, in the section "Other **CDib** Member Functions."

The `biSizeImage` data member of the DIB's `BITMAPINFOHEADER` structure is filled in by calling yet another `CDib` member function, `GetDibSizeImage()`:

```
m_pBmInfoHeader->biSizeImage =  
    GetDibSizeImage();
```

Next, if the `BITMAPINFOHEADER`'s `biClrUsed` member is 0, `LoadBitmapFile()` initializes it to the correct value, which is now stored in the `m_numColors` data member:

```
if (m_pBmInfoHeader->biClrUsed == 0)  
    m_pBmInfoHeader->biClrUsed = m_numColors;
```

Finally, `LoadBitmapFile()` calculates the address of the DIB's image by adding the size of the `BITMAPINFOHEADER` structure and the size of the color table to the `pDib` pointer, which contains the address of the DIB in memory:

```
DWORD clrTableSize = m_numColors * sizeof(RGBQUAD);  
m_pDibBits =  
    pDib + m_pBmInfoHeader->biSize + clrTableSize;
```

## Other **CDib** Member Functions

In the previous section, some `CDib` data members were initialized by calling `CDib` member functions. One of those member functions was `GetDibSizeImage()`, which calculates the size of the DIB in bytes. This function first checks whether the `biSizeImage` member of the `BITMAPINFOHEADER` structure already contains a value other than 0:

```
if (m_pBmInfoHeader->biSizeImage == 0)
```

If it does, the function simply returns the value stored in `biSizeImage`:

```
return m_pBmInfoHeader->biSizeImage;
```

Otherwise, the function must calculate the image size using the information it already has.

First, it gets the DIB's width and height in pixels:

```
DWORD byteWidth = (DWORD) GetDibWidth();  
DWORD height = (DWORD) GetDibHeight();
```

Then it multiplies the width by the height to get the size of the entire bitmap image:

```
DWORD imageSize = byteWidth * height;
```

Notice that all these calculations are done with DWORDs to avoid the truncation errors that might occur with regular integers. DIBs can be big!

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Another member function that must do some calculating is GetDibNumColors() shown in Listing 17.11.

**Listing 17.11 LST17\_11.CPP—Getting the Number of Colors**

```
UINT CDib::GetDibNumColors()
{
    if ((m_pBmInfoHeader->biClrUsed == 0) &&
        (m_pBmInfoHeader->biBitCount < 9))
        return (1 << m_pBmInfoHeader->biBitCount);
    else
        return (int) m_pBmInfoHeader->biClrUsed;
}
```

This function first checks the BITMAPINFOHEADER structure’s biClrUsed member for a value other than 0. If it finds a value other than 0, it simply returns that value from the function. Otherwise, it calculates the number of colors by shifting the value 1 to the left, using the biBitCount member as the shift count. This results in a value of 2 for 1-bit (monochrome) DIBs, 16 for 4-bit DIBs, and 256 for 8-bit DIBs.

The remaining CDib member functions just return the values of the class’s data members. For example, the GetDibRGBTablePtr() function (see Listing 17.12) returns a pointer to the DIB’s color table, a value that has already been stored in the data member m\_pRGBTable.

**Listing 17.12 LST17\_12.CPP—Getting a Pointer to the Color Table**

```
LPRGBQUAD CDib::GetDibRGBTablePtr()
{
    return m_pRGBTable;
}
```

**Creating ShowDib, Version 1**

Now that you have a class for handling DIBs, it’s time to put the class to work. In this section, you’ll use Visual C++’s AppWizard to create an application that can load and display DIBs.

## Creating the Basic Application

In this section, you'll create the ShowDib application, which displays DIBs in its main window. Just follow the steps that come next to create the first version of the application.



The complete source code and executable file for this part of the ShowDib application can be found in the CHAP17\ShowDib, Part 1 directory on this book's CD-ROM.

1. Start Visual C++, and select the File menu's New command. The New property sheet appears, as shown in Figure 17.2.



**FIG. 17.2** The New property sheet enables you to start many types of new projects.

2. In the Project Name edit box, type the project name **ShowDib**, and in the Project Path box, select the directory in which you want to store the project. Finally, make sure that the selected project type in the left-hand pane is MFC AppWizard (exe).
3. Select the OK button. The Step 1 dialog box appears. Choose the Single Document option, as shown in Figure 17.3.



**FIG. 17.3** The Step 1 dialog box determines the basic application type.

4. Click the Next button three times. You see Step 4 of 6 dialog box. Turn off all of the features except Use 3D Controls, as shown in Figure 17.4.
5. Click the Finish button. When the New Project Information dialog box appears (see Figure 17.5), click the OK button to close the dialog box and to generate the files needed for the project.

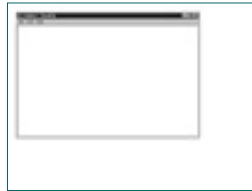


**FIG. 17.4** The Step 4 of 6 dialog box determines what standard features the application will support.



**FIG. 17.5** The New Project Information dialog box displays the project selections you've made.

6. Select the Build, Build command (or press F7) to compile and link the new ShowDib application.
7. To run the program after it's compiled, select the Build, Execute command or press Ctrl+F5 on your keyboard. The window shown in Figure 17.6 appears.



**FIG. 17.6** The basic ShowDib application looks like this.

## Modifying ShowDib's Resources

Now that you have the basic ShowDib application created, you can use Visual C++'s tools to modify the application to fit your needs. The following steps describe how to modify the ShowDib application's resources:

The complete source code and executable file for this part of the ShowDib application can be found in the CHAP17\ShowDib, Part 2 directory on this book's CD-ROM.



1. Click the ResourceView tab. Visual C++ displays the ResourceView window, as shown in Figure 17.7.



**FIG. 17.7** The ResourceView window displays the project's resources.

2. In the ResourceView window, double-click ShowDib Resources, double-click the Menu resource, and then double-click the IDR\_MAINFRAME menu ID. Visual C++'s menu editor appears, as shown in Figure 17.8.



**FIG. 17.8** The menu editor enables you to change the default menus.

3. Click ShowDib's Edit menu (not Visual C++'s Edit menu), and then

press your keyboard's Delete key to delete the Edit menu. When you do, a dialog box asks for verification of the delete command. Click the OK button.

4. Click ShowDib's File menu (not Visual C++'s File menu). The File menu appears, showing its various commands. Using your mouse and your keyboard's Delete key, delete all the File menu's commands except Open and Exit. When you're done, ShowDib's menus should look like Figure 17.9.

5. Close the menu editor, and then double-click the Accelerator resource in the browser window. Double-click the IDR\_MAINFRAME accelerator ID to bring up the accelerator editor, as shown in Figure 17.10.

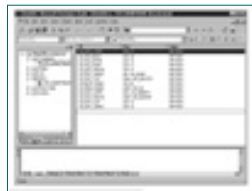
6. Using your keyboard's arrow and Delete keys, delete all accelerators except ID\_FILE\_OPEN, as shown in Figure 17.11.

7. Close the accelerator editor and then double-click the Dialog resource in the browser window. Double-click the IDD\_ABOUTBOX dialog box ID to bring up the dialog box editor.

8. Modify the dialog box by adding the static string "by Macmillan ComputerPublishing", as shown in Figure 17.12.



**FIG. 17.9** Here's what ShowDib's new menus should look like.



**FIG. 17.10** The accelerator editor enables you to modify the default accelerators.

9. Close the dialog box editor. Then select the Build, Build command to compile the modified application.



**FIG. 17.11** This is the accelerator editor after you delete unneeded accelerators.



**FIG. 17.12** Modify the About dialog box so that it looks like this.

When you run the newly compiled application, you see the application's main window. If you select the File menu's Open command, the File Open dialog box appears, from which you can select a file. When you select a file, its name appears in the window in place of the "Untitled" string, as shown in Figure 17.13. If you select the Help menu's About ShowDib& command, you see the About dialog box (also shown in Figure 17.13), which you modified in the dialog box editor.



**FIG. 17.13** Here is ShowDib after you open MAINFRM.CPP and display the About dialog box.

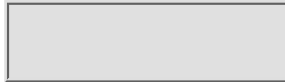
<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Adding Code to ShowDib, Version 1

After you've edited the application's resources, ShowDib's user interface is complete. However, the program is still incapable of loading and displaying DIBs. Even when you use the Open command to select a DIB file, nothing happens (except that the DIB's file name appears in the window's title bar). The next step, then, is to add the code needed to make ShowDib do what you want it to do. The following steps describe how to add this code to the application.

The complete source code and executable file for this part of the ShowDib application can be found in the CHAP17\ShowDib, Part 3 directory on this book's CD-ROM.



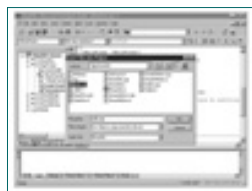
1. Load the MainFrm.cpp file, and then add the following lines to the PreCreateWindow() function, right before the function's return statement:

```
cs.cx = 640;  
cs.cy = 480;
```

2. Copy the cdib.cpp and cdib.h files into ShowDib's project directory. (You'll find cdib.cpp and cdib.h on this book's CD-ROM, in the CHAP17\ShowDib directory.)
3. Select the Project, Add to Project, Files command. The Insert Files into Project dialog box appears, as shown in Figure 17.14. In the file window, double-click the cdib.cpp file to add it to the project.
4. Load the ShowDibDoc.h file, and then add the following line to the top of the source code, right before the declaration of the CShowDibDoc class:

```
#include "cdib.h"
```

This line includes the CDib class's declaration into the ShowDib application's document header file so that you can use the CDib class in the document class.



**FIG. 17.14** Use the Insert Files into Project dialog box to add cdib.cpp to the ShowDib project.

5. In ShowDibDoc.h, add the following line to the Attributes section of the CShowDibDoc class's declaration, right after the public keyword:

```
CDib* m_pDib;
```

This line declares a pointer to a CDib object as a data member of the CShowDibDoc class. The CDib object represents whatever DIB the application is currently displaying.

**6.** Load the ShowDibDoc.cpp file, and then add the following line to the class's constructor, after the // TODO: add one-time construction code here comment:

```
m_pDib = 0;
```

This line ensures that the CDib pointer starts off as NULL.

**7.** Select the View, ClassWizard command. The MFC ClassWizard dialog box appears.

**8.** In the Class Name box, select the CShowDibDoc class. Then click ID\_FILE\_OPEN in the Object IDs box, and double-click COMMAND in the Messages box. The Add Member Function dialog box appears.

**9.** Click the OK button to add the OnFileOpen() function to the class. Click the Edit Code button to open the source code window to this new function.

**10.** Add the code shown in Listing 17.13 to the new OnFileOpen() function, after the // TODO: Add your command handler code here comment.

---

**Listing 17.13 LST17\_13.CPP—Code for the OnFileOpen() Function**

---

```
// Construct an Open dialog-box object.
CFileDialog fileDialog(TRUE, "bmp", "*.bmp");

// Display the Open dialog box.
int result = fileDialog.DoModal();

// If the user exited the dialog box
// via the OK button...
if (result == IDOK)
{
    // Get the selected path and file name.
    CString string = fileDialog.GetPathName();

    // Construct a new CDib object.
    m_pDib = new CDib(string);

    // Check that the CDib object was created okay.
    // If there was an error, the pBmInfo pointer
    // will be 0.
    LPBITMAPINFO pBmInfo = m_pDib->GetDibInfoPtr();

    // If the CDib object was not constructed
    // properly, delete it.
```

```

        if (!pBmInfo)
            DeleteContents();

        // Otherwise, set the document's title to
        // the DIB's path and file name.
        else
            SetTitle(string);
    }

    // Notify the view object that it has new data to display.
    UpdateAllViews(0);

```

---

The `OnFileOpen()` function will now respond to the File menu's Open command, not only by enabling the user to select a file, but also by creating a new `CDib` object from the selected DIB. This function is discussed in detail later in this chapter, in the section “Examining the ***OnFileOpen()*** Function.”

11. Select the View, ClassWizard command. When the MFC ClassWizard dialog box appears, select the `CShowDibDoc` class in the Class Name box; click `CShowDibDoc` in the Object IDs box, and double-click `DeleteContents` in the Messages box to add the virtual `DeleteContents()` function.
12. Click the Edit Code button to jump to the `DeleteContents()` function and then add the code shown in Listing 17.14 to the function, after the `// TODO: Add your specialized code here and/or call the base class comment`.

---

#### Listing 17.14 LST17\_14.CPP—Code for the `DeleteContents()` Function

---

```

// If there's a valid CDib object, delete it.
if (m_pDib)
{
    delete m_pDib;
    m_pDib = 0;
}

```

---

The `DeleteContents()` function overrides the `CDocument` base class's `DeleteContents()` function. MFC calls this function whenever the application needs to delete the current document's data. Because `DeleteContents()` is also called when a new document is created (to ensure that the new document is empty), you must check that `m_pDib` is not 0 before you delete it.

13. Open the `ShowDibView.cpp` file, and then add the code shown in Listing 17.15 to the `OnDraw()` function, right after the `// TODO: add draw code for native data here comment`.

---

#### Listing 17.15 LST17\_15.CPP—Code for the `OnDraw()` Function

---

```

// Get a pointer to the current CDib object.
CDib* pDib = pDoc->m_pDib;

```



```
// If the CDib object is valid, display it.
if (pDib)
{
    // Get a pointer to the DIB's image data.
    BYTE* pBmBits = pDib->GetDibBitsPtr();

    // Get a pointer to the DIB's info structure.
    LPBITMAPINFO pBmInfo = pDib->GetDibInfoPtr();

    // Get the DIB's width and height.
    UINT bmWidth = pDib->GetDibWidth();
    UINT bmHeight = pDib->GetDibHeight();

    // Display the DIB.
    StretchDIBits(pDC->m_hDC,
        10, 10, bmWidth, bmHeight,
        0, 0, bmWidth, bmHeight,
        pBmBits, pBmInfo, DIB_RGB_COLORS, SRCCOPY);
}
```

---

MFC calls the OnDraw() function whenever the application's main window must be redrawn. The code in Listing 17.15, which is described in detail later in the section "Examining the *OnDraw()* Function," displays the currently loaded DIB.

**14.** Select the Build, Build command to compile and link the new version of the ShowDib application.

Version 1 of ShowDib is now complete. Before you run the application, however, read the following sections, which explain how the OnFileOpen() and OnDraw() functions work.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Examining the *OnFileOpen()* Function

As you put together the first version of the ShowDib application, you encountered a couple of functions that needed further explanation. One of those functions is OnFileOpen().

Normally, an application created by AppWizard responds to the File menu's Open command by displaying an Open dialog box and then calling the document class's Serialize() function, which loads the requested data. You saw this happen when you ran the basic version of ShowDib.

Unfortunately, because ShowDib requires some special file handling, it doesn't take advantage of the Serialize() function. Instead, the program uses the OnFileOpen() function to load a DIB. Because you "attached" this function to the File menu's Open command, MFC calls it whenever a file needs to be opened.

The function's first task is to display the Open dialog box so that the user can choose the file that he or she wants to view:

```
CFileDialog fileDialog(TRUE, "bmp", "*.bmp");
```

MFC encapsulates the Open dialog box in the CFileDialog class. To construct a CFileDialog object, call its constructor with three arguments: the Boolean value TRUE, a string containing the default file extension, and a string containing the initial file name to appear.

After you've constructed a new CFileDialog object, display the dialog box by calling the class's DoModal() member function:

```
int result = fileDialog.DoModal();
```

The function DoModal() returns the ID of the button used to exit the dialog box. If that ID equals ID\_OK, the user has selected a valid file name. In that case, OnFileOpen() gets the selected file name by calling the file dialog's GetPathName() member function:

```
CString string = fileDialog.GetPathName();
```

The program then uses the returned file name to construct a new CDib object, which, as you already know, also loads the DIB into memory. If CDib's constructor cannot load the DIB, it sets all of the CDib object's data members to 0. So to check whether the DIB was loaded correctly, OnFileOpen() calls the

CDib object's GetDibInfoPtr() function to get a pointer to the DIB's BITMAPINFO structure:

```
LPBITMAPINFO pBmInfo = m_pDib->GetDibInfoPtr();
```

If the returned pointer is 0, the DIB was not loaded, and the program calls DeleteContents() to delete the unusable CDib object:

```
if (!pBmInfo)
    DeleteContents();
```

Otherwise, the program calls the document object's SetTitle() member function to set the window's title to the DIB's file name:

```
SetTitle(string);
```

SetTitle()'s single argument is a string containing the window's new title; in this case, the string returned from the dialog object's GetPathName() function.

Because the window now has a new DIB to display, OnFileOpen() calls the document object's UpdateAllViews() member function, to tell the view object that it must redraw the window:

```
UpdateAllViews(0);
```

This function call's single parameter is a pointer to the view object that modified the document. Because, in this case, the document itself changed its contents, this argument should be 0, which indicates that all views should be updated.

## Examining the *OnDraw()* Function

Another function that you need to look at in more detail is the view object's OnDraw() function, which is responsible for updating the main window's display. In the ShowDib application, OnDraw() must display the currently selected DIB.

Because the CDib object is part of the document class, OnDraw() first gets a pointer to the CDib object, accessing the document object through the pDoc pointer supplied by OnDraw(). (The code that supplies this pointer was generated by AppWizard.) As you can see, pDoc points to the application's document object:

```
CDib* pDib = pDoc->m_pDib;
```

If the application has loaded a DIB (pDib is not 0), OnDraw() gets pointers to the DIB's image and BITMAPINFO structure:

```
BYTE* pBmBits = pDib->GetDibBitsPtr();
LPBITMAPINFO pBmInfo = pDib->GetDibInfoPtr();
```

The function also gets the DIB's width and height:

```
UINT bmWidth = pDib->GetDibWidth();  
UINT bmHeight = pDib->GetDibHeight();
```

The program then displays the DIB by calling the Windows function StretchDIBits():

```
StretchDIBits(pDC->m_hDC, 10, 10, bmWidth, bmHeight,  
             0, 0, bmWidth, bmHeight, pBmBits, pBmInfo,  
             DIB_RGB_COLORS, SRCCOPY);
```

This function requires a whopping 13 arguments. The first argument is a handle to the destination device context (that is, the device context on which the DIB should be displayed). Because OnDraw() supplies a pointer to the destination device context (DC), you must use this pointer to access the DC's handle, which is stored in the DC class's m\_hDC data member.

---

**Note:**

To display data, a Windows application must obtain a device context. In review, a *device context* is little more than a collection of attributes that describe how data should be displayed in a window. These attributes include pen colors and sizes, brush colors and sizes, fill colors, and other important graphical information.

In a normal Windows program, you'd be responsible for creating and managing a device context whenever you wanted to display something on the screen. However, Microsoft Foundation Classes takes care of much of the dirty work for you. In functions like OnDraw(), MFC supplies the device context as one of the function's parameters.

---

The next four arguments are the X coordinate, Y coordinate, width, and height of the destination rectangle. This rectangle is the area of the destination DC into which you want to copy the DIB. The next four arguments define the source rectangle. In this case, because you want to display the entire DIB, this rectangle has X,Y coordinates of 0,0 and is the same width and height as the DIB.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The ninth and tenth arguments are pointers to the DIB's image data and BITMAPINFO structure. The eleventh argument is a flag that describes the colors used with the DIB. The flag DIB\_RGB\_COLORS means that the DIB's color table contains actual RGB values, whereas the DIB\_PAL\_COLORS flag indicates that the color table contains 16-bit indexes into the logical palette. (You learn about logical palettes later in this chapter, in the section "Creating ShowDib, Version 2.") Finally, the last argument is a flag that tells the function how to combine the source and destination rectangles. In this case, SRCCOPY means that the source rectangle should overwrite the destination rectangle.

#### Note:

The StretchDIBits() function can automatically stretch or shrink the size of an image when the size of the destination surface is larger or smaller than the bitmap.

Now that you know how the program works, it's time to try it out.

## Running Version 1 of ShowDib

The ShowDib application can now load and display a DIB. To use the program, select the File menu's Open command and select a DIB file. ShowDib loads the selected DIB and displays it in its main window. Figure 17.15 shows the results when you select the DOGGIE2.BMP file, which came from Microsoft's WinG Developer's Kit and is also found on this book's CD-ROM, in the CHAP17\ShowDib directory.



When using this first version of ShowDib, you'll probably notice that, although the program can load and display a DIB, the resulting display sometimes looks pretty weird. This is because, currently, ShowDib ignores the DIB's color table. To display a DIB correctly, a program must create a logical palette from the DIB's color table and then select that palette before actually updating the display window. In the next section, then, you'll add palette-handling features to ShowDib so that it can display DIBs using the correct colors.



## Creating ShowDib, Version 2

Currently, ShowDib cannot display a DIB properly because it does not set Windows' colors to those that were used to create the DIB. The colors that you need are found in the DIB's color palette. To properly display the DIB, you must create a logical palette from the DIB's color table and then *realize* that palette before displaying the DIB. When you realize a palette, you're telling Windows to change its colors to those that you've defined in the logical palette.

But what exactly is a logical palette? Every application that must define colors has its own logical palette. When the user switches between applications, the new application gives its logical palette to Windows so that Windows can set its colors properly. A *logical palette*, then, is a set of colors that's associated with a particular application. The *system palette*, on the other hand, contains the colors that are currently displayed on the screen. When a user switches applications, the new application's logical palette is mapped into Windows' system palette. You can have many logical palettes, but there is never more than one system palette.

The process by which Windows maps a logical palette into the system palette, however, is complex and is a cause for much confusion. That's why you see so little about palettes in most Windows programming books. But, as you might have guessed, if you are going to display DIBs, you must know how Windows' palettes work.

### Mapping Between a Logical and a System Palette

When an application realizes a logical palette, Windows must map the application's palette to the system palette. Because there is only one system palette—and because that system palette must be shared by every currently running application—mapping a logical palette is not as simple as copying the logical palette to the system palette. Although taking this easy way out would work well for the active application, any applications running in the background might end up with bizarre displays. When Windows maps palettes, it must do its best to keep every application's display looking as good as possible. Windows follows these steps to set an application's requested colors:

1. *Exactly match colors in the logical palette to existing colors in the system palette.* This step requires that no colors in the system palette be modified, which, of course, also ensures that other applications retain their own colors.
2. *Use empty system palette entries to create colors for those logical palette entries that could not be matched in Step 1.* By using empty system palette entries (entries not being used by any other application), Windows again ensures that colors used by other applications remain unaffected.
3. *Match remaining entries in the logical palette to the closest color in the system palette.* Obviously, with many applications vying for colors,

the system palette might become filled. When this happens, Windows can no longer plug colors from a logical palette into empty system palette entries. At this point, a background application's display suffers some degradation, depending on how closely the selected colors match the application's palette.

Although Windows follows the preceding rules when mapping colors, the order in which each application is handled is obviously important. For this reason, Windows maps the active application's colors first, after which it maps background applications' colors starting with the most recently active and working its way back to the least recently active.

Need an example? For the sake of simplicity, suppose that the system palette can display only ten colors. Suppose also that you have two applications running—App1 and App2—each of which has realized a six-color palette. Before any mapping occurs, the palettes look like Figure 17.16.

App1 is the active application, so Windows maps App1's colors first. Windows first maps App1's black to the system palette's black because they are an exact match. Then Windows fills the next five empty system palette entries with the five remaining colors in App1's logical palette. After App1's logical palette has been fully realized, the palettes look like Figure 17.17.

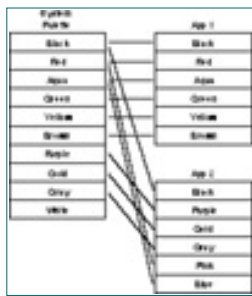
Because App2 is running in the background, Windows maps App2's colors after App1's. Again, Windows maps black to black because it's an exact match. Next, Windows fills the system palette's remaining three empty entries with the next three colors in App2's palette. Now the system palette is filled. Unfortunately, App2 still has two colors—pink and blue—that need to be mapped. Windows now has no choice but to map these two remaining colors to the closest colors in the system palette. So Windows matches pink to red and blue to aqua. The palettes now look like Figure 17.18.



**FIG. 17.16** The unmapped palettes look like this.



**FIG. 17.17** Here are the palettes after mapping App1's palette.



**FIG. 17.18** Here are the palettes after mapping App2's palette.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

After both applications' palettes have been mapped, App1 has exactly the colors it needs, whereas App2 has suffered a slight degradation in its display. But because App2 is currently running in the background, its display isn't as important as App1's. When App2 becomes active, it will get its chance to create the display that it needs.

Now that you know how Windows handles color palettes, it's time to add palette-handling capabilities to the ShowDib application. You'll do this in the next section.

---

#### Note:

The Windows system palette reserves the first ten and last ten colors of a 256-color palette for its own use. It uses these colors to draw window borders, buttons, menus, and other Windows objects. Normally, you should not attempt to change these 20 reserved colors.

---

## Adding Code to ShowDib, Version 2

You're now ready to add palette handling to the ShowDib application. When you're finished with this section, ShowDib will display DIBs correctly, using the DIB's color table to create a logical palette for the application. To add these capabilities, perform the following steps.

The complete source code and executable file for this part of the ShowDib application can be found in the CHAP17\ShowDib, Part 4 directory on this book's CD-ROM.



1. Add the following function declaration to the CShowDibView class, found in the file ShowDibView.h. Place the code in the class's Implementation section, right before the // Generated message map functions comment and right after the protected keyword.

```
HPALETTE CreatedibPalette(CDib* pDib);
```

The preceding line declares the function CreateDibPalette()—which returns a handle to a logical palette—as a protected member function of the CShowDibView class.

2. Add the function shown in Listing 17.16 to the view's implementation file. Place the code at the end of the ShowDibView.cpp file, right after the // CShowDibView message handlers comment.

### Listing 17.16 LST17\_16.CPP—The CreateDibPalette() Function

---

```

HPALETTE CShowDibView::CreateDibPalette(CDib* pDib)
{
    // Get a pointer to the DIB's color table.
    LPRGBQUAD pColorTable = pDib->GetDibRGBTablePtr();

    // Get the number of colors in the DIB.
    UINT numColors = pDib->GetDibNumColors();

    struct
    {
        WORD Version;
        WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
    } logicalPalette = { 0x300, 256 };

    // Fill the palette structure with the DIB's colors.
    for(UINT i=0; i<numColors; ++i)
    {
        logicalPalette.aEntries[i].peRed =
            pColorTable[i].rgbRed;
        logicalPalette.aEntries[i].peGreen =
            pColorTable[i].rgbGreen;
        logicalPalette.aEntries[i].peBlue =
            pColorTable[i].rgbBlue;
        logicalPalette.aEntries[i].peFlags = 0;
    }

    // Create the palette object.
    HPALETTE hPalette =
        CreatePalette((LPLOGPALETTE)&logicalPalette);

    return hPalette;
}

```

---

This function creates the application's logical palette based on the DIB's color table. You'll examine this function in greater detail later in this chapter, in the section "Examining the *CreateDibPalette()* Function."

3. Add the code shown in Listing 17.17 to the view's OnDraw() function. Place the code after the if statement's opening brace.

---

**Listing 17.17 LST17\_17.CPP—Code for the OnDraw() Function**

---

```

// Create the DIB's palette;
HPALETTE hPalette = CreateDibPalette(pDib);

// Select the DIB's palette into the DC.

```

```
HPALETTE hOldPalette =  
    SelectPalette(pDC->m_hDC, hPalette, FALSE);  
  
// Map the DIB's palette to the system palette.  
RealizePalette(pDC->m_hDC);
```

---

The code in Listing 17.17 creates and realizes the application's logical palette before displaying the DIB. You examine this code in greater detail later in this chapter, in the section “Examining Changes to the *OnDraw()* Function.”

4. Add the code shown in Listing 17.18 immediately after the StretchDIBits() call in OnDraw().

---

#### **Listing 17.18 LST0R\_18.CPP—More Code for the OnDraw() Function**

---

```
// Deselect the logical palette from the DC.  
SelectPalette(pDC->m_hDC, hOldPalette, FALSE);  
  
// Delete the logical palette.  
DeleteObject(hPalette);
```

---

This code deletes the logical palette. Later in this chapter, in the section “Examining Changes to the *OnDraw()* Function,” you’ll see why deleting the palette is necessary.

Version 2 of ShowDib is now complete. Before you run the application, however, read the following sections, which explain how the new code that you added works.

### **Examining the *CreateDibPalette()* Function**

The CreateDibPalette() function that you just added to the application's view class creates a logical palette from the currently loaded DIB's color table. In that function, the program's first task is to obtain a pointer to the CDib object's color table. It does this by calling the CDib object's GetDibRGBTablePtr() member function:

```
LPRGBQUAD pColorTable = pDib->GetDibRGBTablePtr();
```

CreateDibPalette() also needs to know how many colors there are in the DIB's color table. It gets this value by calling the CDib object's GetDibNumColors() member function:

```
UINT numColors = pDib->GetDibNumColors();
```

Next, CreateDibPalette() defines a structure, as shown in Listing 17.19, to hold the information that Windows needs to create a logical palette.

---

#### **Listing 17.19 LST17\_19.CPP—The Logical Palette Structure**

---

```
struct
```

```
{
    WORD Version;
    WORD NumberOfEntries;
    PALETTEENTRY aEntries[256];
} logicalPalette = { 0x300, 256 };
```

---

The lines in Listing 17.19 represent a LOGPALETTE structure, a data type defined by Windows and used when handling logical palettes. The first structure member is a version number, which should be the hex value 0x300. The second member is the number of colors in the palette. As you can see, CreateDibPalette() creates a 256-color logical palette. The final member is an array of PALETTEENTRY structures. Windows defines the PALETTEENTRY structure as shown in Listing 17.20.

#### **Listing 17.20 LST17\_20.CPP—The PALETTEENTRY Structure**

---

```
typedef struct {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

---

The peRed, peGreen, and peBlue members of this structure hold the intensities of the color's red, green, and blue components. The peFlags member specifies how Windows should handle the palette entry and can be the values PC\_EXPLICIT, PC\_NOCOLLAPSE, PC\_RESERVED, or 0.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The PC\_EXPLICIT flag indicates that the palette entry contains an index into the system palette, rather than actual color values. You store this index in the low-order word of the entry. You'll rarely need to use the PC\_EXPLICIT flag.

The PC\_NOCOLLAPSE flag specifies that Windows should place the color into an empty system-palette entry, rather than map it to an existing entry. If there are no empty entries in the system palette, Windows overrides the PC\_NOCOLLAPSE flag and performs normal matching instead. Other applications might map their own colors to a palette entry marked PC\_NOCOLLAPSE.

The PC\_RESERVED flag prevents Windows from matching other applications' colors to the color entry. You'd usually use the PC\_RESERVED flag with animated palettes, which frequently change color.

Set the peFlags member to 0 when you want to let Windows handle the color entry any way it sees fit.

Getting back to CreateDibPalette(), the function next copies the DIB's color table into the logicalPalette structure and sets the peFlags member of each entry to 0, as shown in Listing 17.21.

#### Listing 17.21 LST17\_21.CPP—Initializing the Logical Palette Structure

```
for(UINT i=0; i<numColors; ++i)
{
    logicalPalette.aEntries[i].peRed =
        pColorTable[i].rgbRed;
    logicalPalette.aEntries[i].peGreen =
        pColorTable[i].rgbGreen;
    logicalPalette.aEntries[i].peBlue =
        pColorTable[i].rgbBlue;
    logicalPalette.aEntries[i].peFlags = 0;
}
```

Finally, CreateDibPalette() creates the logical palette by calling the Windows API function CreatePalette(), after which it returns a handle to the palette:

```
hPalette =
    CreatePalette((LPLOGPALETTE)&logicalPalette);
return hPalette;
```

The Windows API function `CreatePalette()`, which returns a handle to a logical palette, takes as its single argument a pointer to a `LOGPALETTE` structure.

## Examining Changes to the *OnDraw()* Function

In `ShowDib`'s `OnDraw()` function, you just added several lines of code that you might not understand. This section describes more fully what that new code accomplishes.

The first line that you added simply calls your new function, `CreateDibPalette()`, to create a logical palette from the DIB's color table:

```
HPALETTE hPalette = CreateDibPalette(pDib);
```

The next line selects the newly created logical palette into the device context:

```
HPALETTE hOldPalette =  
    SelectPalette(pDC->m_hDC, hPalette, FALSE);
```

The Windows API function `SelectPalette()` selects a palette into a DC. Its three arguments are a handle to the DC, a handle to the palette, and a Boolean value indicating whether the palette is to always be a background palette (`TRUE`) or whether the palette should be a foreground palette when the window is active (`FALSE`). You'll usually use `FALSE` for this argument. Note that `SelectPalette()` returns a handle to the old palette. You need to save this handle so that later you can remove the palette that you create.

After `OnDraw()` selects the logical palette into the DC, it must realize the palette, which tells Windows to map the palette to the system palette. The next line does this by calling the Windows API function `RealizePalette()`:

```
RealizePalette(pDC->m_hDC);
```

This function takes as its single argument the handle of the device context whose palette should be realized.

After realizing the logical palette, `OnDraw()` displays the DIB, after which it deletes the logical palette. However, because an application must never delete an object that's selected into a DC, `OnDraw()` must first remove the logical palette from the DC. It does this by selecting the old palette back into the DC:

```
SelectPalette(pDC->m_hDC, hOldPalette, FALSE);
```

Then the program can safely delete the logical palette that it created:

```
DeleteObject(hPalette);
```

---

### Note:

Why delete a logical palette? Because if you don't delete graphical objects that you create under Windows 3.1, they continue to take up memory after your application ends. If your application causes enough of this type of *memory leak*, you could prevent other applications from running correctly.

However, although it's always good practice to delete allocated objects, this hard-and-fast rule really applies only to 16-bit applications or 32-bit applications running under Win32s in Windows 3.1. Because 32-bit applications running in Windows 95 or Windows NT have their own address spaces, undeleted objects don't affect other applications in the system.

---

## Running Version 2 of ShowDib

The ShowDib application can now load and display a DIB, using the correct color palette. To use the program, select the File menu's Open command and select a DIB file. ShowDib loads the selected DIB and displays it in its main window. Figure 17.19 shows the results when you select the DOGGIE2.BMP file. As you see, ShowDib's display looks much better when the program handles palettes correctly.



**FIG. 17.19** Version 2 of ShowDib now displays bitmaps with correct palette handling.

Unfortunately, although ShowDib now displays DIBs properly when it's the active application, it sometimes produces less than desirable results when it's running in the background. For example, Figure 17.20 shows ShowDib running in the background while PC Paintbrush is running in the foreground with a different 256-color palette from the one that ShowDib needs to properly display the DOGGIE2.BMP DIB. Ouch!



**FIG. 17.20** This figure shows version 2 of ShowDib running in the background without responding to the WM\_PALETTECHANGED message.

Why isn't ShowDib correctly updating its display? Shouldn't Windows be mapping colors such that ShowDib produces a better looking display than that shown in Figure 17.20? The truth is that Windows is doing everything that it can. The problem is ShowDib, which is not responding to Windows' instructions. Specifically, ShowDib is not responding to WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED messages. You'll learn about these messages as you create version 3 of the ShowDib application.

## Creating ShowDib, Version 3

Version 2 of ShowDib enables you to load a DIB and display it correctly. But ShowDib does not yet behave correctly when it's relegated to background operation. Its display can become badly corrupted when the user activates

another palette-managed application. Version 3 of ShowDib handles this problem by responding to two important Windows messages, WM\_PALETTECHANGED and WM\_QUERYNEWPALETTE.

## Responding to Palette Messages

When the user activates an application, that application might realize its own logical palette. When this happens, the colors in the system palette might change significantly, leaving background applications looking strange. The solution to this problem is to notify background applications that the system palette has changed, which Windows does by sending a WM\_PALETTECHANGED message.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[Brief](#)   [Full](#)  
 • [Advanced Search](#)  
 • [Search Tips](#)

 **BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

When a background application receives this message, it can realize its own logical palette, which tells Windows to remap the logical palette to the system palette. Because the active application has first claim on the colors in the system palette, background applications might lose some of their display integrity even when responding to WM\_PALETTECHANGED, but at least Windows will try for the best match possible between each background application's logical palette and the new system palette.

When a background application is reactivated, it can reclaim the system palette. Windows notifies the application of this opportunity by sending a WM\_QUERYNEWPALETTE message. The newly activated application responds to this message by realizing its logical palette, which forces all of the logical palette's colors into the system palette. At this point, the other running applications, which are now background applications, receive WM\_PALETTECHANGED messages; then they can remap their logical palettes to the system palette, which you just changed.

In the following section, you'll learn how to modify a Visual C++ program to respond to the WM\_PALETTECHANGED and WM\_QUERYNEWPALETTE messages.

### Adding Code to ShowDib, Version 3

To complete the ShowDib application, giving it the capability to update its display properly when it's running in the background, follow the steps listed next.



The complete source code and executable file for this final part of the ShowDib application can be found in the CHAP17\ShowDib directory on this book's CD-ROM.

1. Select the View, ClassWizard command. When the MFC ClassWizard dialog box appears, select CMainFrame in the Class Name list box.
2. Select CMainFrame in the Object IDs list box, and then double-click the WM\_PALETTECHANGED and WM\_QUERYNEWPALETTE messages in the Messa\_ges list box. ClassWizard adds the OnPaletteChanged() and OnQueryNewPalette() member functions to the CMainFrame class. Click the OK button to finalize your choices.
3. Click the Edit Code button to open the MainFrm.cpp file to the new OnPaletteChanged() and OnQueryNewPalette() functions.
4. Add the code shown in Listing 17.22 to the OnPaletteChanged()

function right after the // TODO: Add your message handler code here comment.

---

**Listing 17.22 LST17\_22.CPP—Code for the OnPaletteChanged() Function**

---

```
// Get a pointer to the view window.
CView* pView = GetActiveView();

// If it isn't the view window changing
// the palette, redraw the view window with
// the newly mapped palette.
if (pFocusWnd != pView)
    pView->Invalidate();
```

---

This code, which you'll examine in detail later in this chapter, in the section “Examining the *OnPaletteChanged()* Function,” notifies the application's view that it must realize the logical palette and redraw the display.

5. Add the code shown in Listing 17.23 to the OnQueryNewPalette() function, right after the // TODO: Add your message handler code here and/or call default comment.

---

**Listing 17.23 LST17\_23.CPP—Code for the OnQueryNewPalette() Function**

---

```
// Get a pointer to the view window.
CView* pView = GetActiveView();

// Redraw the view window with its own colors.
pView->Invalidate();
```

---

This code, which you'll examine in detail in the section “Examining the *OnQueryNewPalette()* Function,” notifies the application's view that it must realize the logical palette and redraw the display.

Version 3 of ShowDib is now complete. Before you run the application, however, read the following sections, which explain how the new code that you added works.

### Examining the *OnPaletteChanged()* Function

The OnPaletteChanged() function is called whenever Windows sends the application a WM\_PALETTECHANGED message, which it does whenever another application changes the system palette. When ShowDib receives this message, it must realize its palette and redraw its display. Because the view class is responsible for updating the display, OnPaletteChanged() must first acquire a pointer to the view:

```
CView* pView = GetActiveView();
```

Next, the program must check whether it was its own view that modified the system palette. If it was, `OnPaletteChanged()` should ignore the `WM_PALETTECHANGED` message. Failure to handle the message in this way will leave your application in an infinite loop as it continually realizes its palette and then responds to the resulting `WM_PALETTECHANGED` message. Because the `OnPaletteChanged()` function receives as its single parameter a pointer to the window that changed the palette, you can compare this pointer with a pointer to your application's view:

```
if (pFocusWnd != pView)
```

If the two pointers are not equal, some other application changed the palette. In this case, `OnPaletteChanged()` calls the view's `Invalidate()` member function, which forces a call to the view's `OnDraw()` function:

```
pView->Invalidate();
```

In `OnDraw()`, the program realizes its palette and draws the currently selected DIB. By realizing its palette in response to the `WM_PALETTECHANGED` message, the program's logical palette is remapped to the newly changed system palette, creating the best possible display for this application as it's running in the background.

## Examining the *OnQueryNewPalette()* Function

The `OnQueryNewPalette()` function is called whenever Windows sends the application a `WM_QUERYNEWPALETTE` message, which it does whenever the application regains the focus (becomes the top window). Just as with the `WM_PALETTECHANGED` message, when `ShowDib` receives the `WM_QUERYNEWPALETTE` message, it must realize its palette and redraw its display. Because the view class is responsible for updating the display, `OnQueryNewPalette()` must first acquire a pointer to the view:

```
CView* pView = GetActiveView();
```

`OnQueryNewPalette()` then calls the view's `Invalidate()` member function, which forces a call to the view's `OnDraw()` function:

```
pView->Invalidate();
```

In `OnDraw()`, the program realizes its palette and draws the currently selected DIB. By realizing its palette in response to the `WM_QUERYNEWPALETTE` message, the program's logical palette is remapped to the system palette. Because `ShowDib` is, at this point, the active application, it can take over the system palette and create exactly the palette that it needs.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 18.1 Member Functions of the Array Classes

Function	Description
Add()	Appends a value to the end of the array, increasing the size of the array as needed.
ElementAt()	Gets a reference to an array element's pointer.
FreeExtra()	Releases unused array memory.
GetAt()	Gets the value at the specified array index.
GetSize()	Gets the number of elements in the array.
GetUpperBound()	Gets the array's <i>upper bound</i> , which is the highest valid index at which a value can be stored.
InsertAt()	Inserts a value at the specified index, shifting existing elements upward as necessary to accommodate the insert.
RemoveAll()	Removes all of the array's elements.
RemoveAt()	Removes the value at the specified index.
SetAt()	Places a value at the specified index. Because this function will not increase the size of the array, the index must be currently valid.
SetAtGrow()	Places a value at the specified index, increasing the size of the array as needed.
SetSize()	Sets the array's size, which is the number of elements that the array can hold. The array still can grow dynamically beyond this size.

Introducing the Array Demo Application

To illustrate how the array classes work, this chapter includes the Array Demo application, which you can find in the Chap18\array folder of this book's CD-ROM. When you run the program, you see the window shown in Figure 18.1. The window displays the current contents of the array. Because the application's array object (which is an instance of CUIntArray) starts off with ten elements, the values for these elements (indexed as 0 through 9) are displayed on the screen. The application enables you to change, add, or delete elements in the array and see the results.



FIG. 18.1 The Array Demo application lets you experiment with MFC's array classes.

You can add an element to the array in several ways. To see these choices, left-click in the application's

window. The dialog box shown in Figure 18.2 appears. Type an array index in the Index box and the new value in the Value box. Then select whether you want to set, insert, or add the element. When you choose Set, the element that you specify in the Index field gets changed to the value in the Value field.

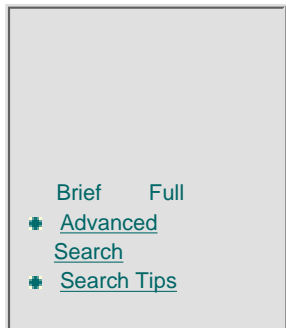
The insert operation creates a new array element at the location specified by the index, pushing succeeding elements forward. Finally, the Add operation just tacks the new element onto the end of the array. In this case, the program ignores the Index field of the dialog box.



**FIG. 18.2** The Add to Array dialog box lets you add elements to the array.

[Previous](#) [Table of Contents](#) [Next](#)

 **SEARCH**  
ITKNOWLEDGE



 **BROWSE**  
BY TOPIC



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 18

# Using the MFC Collection Classes

- How to create array, list, and map collection objects
- How to add, remove, and modify collection elements
- How to iterate over collections
- How to create your own type of collection class

MFC includes a lot more than classes for programming Windows' graphical user interface. It also features many collection classes for handling such things as lists, arrays, and maps. By using these classes, you gain extra power over data in your programs, and you simplify many operations involved in using complex data structures such as lists.

For example, because MFC's *array classes* can change their size dynamically, you are relieved of creating oversized arrays in an attempt to ensure that the arrays are large enough for the application. In this way, you save memory. The other collection classes provide many similar conveniences. In this chapter, you take a close look at MFC's collection classes.

## The Array Classes

MFC's array classes enable you to create and manipulate one-dimensional array objects that can hold virtually any type of data. These array objects work much like the standard arrays that you're accustomed to using in your

programs, except that MFC can enlarge or shrink an array object dynamically at runtime. This means that you don't have to be concerned with dimensioning your array perfectly when it's declared. Because MFC's arrays can grow dynamically, you can forget about the memory waste that often occurs with conventional arrays, which must be dimensioned to hold the maximum number of elements that might be needed in the program, whether or not you actually use every element.

The array classes include CByteArray, CDWordArray, CObArray, CPtrArray, CUIntArray, CWordArray, and CStringArray. As you can tell from the class names, each class is designed to hold a specific type of data. For example, the CUIntArray, which is used in this section's examples, is an array class that can hold unsigned integers. The CPtrArray class, on the other hand, represents an array of pointers to void, and the CObArray class represents an array of objects. The array classes are almost identical, differing only in the type of data that they store. As a result, once you've learned to use one of the array classes, you've learned to use them all. Table 18.1 lists the member functions of the array classes and their descriptions.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Suppose, for example, that you enter 3 into the dialog box’s Index field and 15 into the Value field, leaving the option buttons set to Set. Figure 18.3 shows the result, where the program has placed the value 15 into element 3 of the array, overwriting the value that was there previously. Now say that you type 5 into Index, 25 into Value, and click the Insert option button.



**FIG. 18.3** The value of 15 has been placed into array element 3.

In Figure 18.4, you can see that the program stuffs a new element 5 into the array, shoving the other elements forward. The Add option button tells the program to add a new element to the end of the array.



**FIG. 18.4** The screen now shows the new array element 5, giving 11 elements in all.

An interesting thing to try—something that really shows how dynamic MFC’s arrays are—is to set an array element beyond the end of the array. For example, given the program’s state shown in Figure 18.4, if you type 20 in Index and 45 in Value and then choose the Set option button, you get the results shown in Figure 18.5. Because there was no element 20, the array class created the new elements that it needed to get to 20. Try that with an old-fashioned array!



**FIG. 18.5** The array class has added the elements needed to set element 20.

Besides adding new elements to the array, you can also delete elements in one of two ways. To do this, first right-click the window. When you do, you see the dialog box shown in Figure 18.6. If you type an index into the Remove field and then click OK, the program deletes the selected element from the array. This is the opposite of the effect of the Insert command, because the Remove command shortens the array rather than lengthens it. If you want, you can select the Remove All option in the dialog box. Then the program deletes all elements from the array, leaving it empty.





**FIG. 18.6** The Remove from Array dialog box lets you delete elements from the array.

## Declaring and Initializing the Array

Now you'd probably like to see how all this array trickery works. It's really pretty simple. First, the program declares the array object as a data member of the view class, like this:

```
CUIntArray array;
```

Then, in the view class's constructor, the program initializes the array to ten elements:

```
array.SetSize(10, 5);
```

The `SetSize()` function takes as parameters the number of elements to give the array initially and the number of elements by which the array should grow whenever it needs to. You don't need to call `SetSize()` to use the array class. However, if you fail to do so, MFC adds elements to the array one at a time, as needed, which is a slow process (although, unless you're doing some heavy processing, you're not likely to notice any difference in speed). By giving an initial array size and the amount by which to grow, you can create much more efficient array-handling code.

## Adding Elements to the Array

After setting the array size, the program waits for the user to click the left or right mouse buttons in the window. When the user does, the program springs into action, displaying the appropriate dialog box and processing the values entered into the dialog box. Listing 18.1 shows the Array Demo application's `OnLButtonDown()` function, which handles the left mouse button clicks.

### Listing 18.1 LST18\_01.cpp—The `OnLButtonDown()` Function

---

```
void CArrayView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    ArrayDlg dialog(this);
    dialog.m_index = 0;
    dialog.m_value = 0;
    dialog.m_radio = 0;
    int result = dialog.DoModal();
    if (result == IDOK)
    {
        if (dialog.m_radio == 0)
            array.SetAtGrow(dialog.m_index, dialog.m_value);
        else if (dialog.m_radio == 1)
            array.InsertAt(dialog.m_index, dialog.m_value, 1);
        else
            array.Add(dialog.m_value);
        Invalidate();
    }
    CView::OnLButtonDown(nFlags, point);
}
```

---

If the user exits the dialog box by clicking the OK button, the `OnLButtonDown()` function checks the value of the dialog box's `m_radio` data member. A value of 0 means that the first option button (Set) is set; 1 means that the second button (Insert) is set; and 2 means that the third button (Add) is set.

If the user wants to set an array element, the program calls `SetAtGrow()`, giving the array index and the new value as arguments. Unlike the regular `SetAt()` function, which you can use only with a currently valid index number, `SetAtGrow()` will enlarge the array as necessary to set the specified array element.

When the user has selected the Insert option button, the program calls the `InsertAt()` function, giving the array index and new value as arguments. This causes MFC to create a new array element at the index specified, shoving the other array elements forward. Finally, when the user has selected the Add option, the program calls the `Add()` function, which adds a new element to the end of the array. This function's single argument is the new value to place in the added element. The call to `Invalidate()` forces the window to redraw the data display with the new information.

## Reading Through the Array

So that you can see what's happening as you add, change, and delete array elements, the Array Demo application's `OnDraw()` function reads through the array, displaying the values that it finds in each element. The code for this function is shown in Listing 18.2.

### Listing 18.2 LST18\_02.cpp—Array Demo's `OnDraw()` Function

---

```
void CArrayView::OnDraw(CDC* pDC)
{
    CArrayDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here

    // Get the current font's height.
    TEXTMETRIC textMetric;
    pDC->GetTextMetrics(&textMetric);
    int fontHeight = textMetric.tmHeight;
    // Get the size of the array.
    int count = array.GetSize();
    int displayPos = 10;
    // Display the array data.
    for (int x=0; x<count; ++x)
    {
        UINT value = array.GetAt(x);
        char s[81];
        wsprintf(s, "Element %d contains the value %u.", x, value);
        pDC->TextOut(10, displayPos, s);
        displayPos += fontHeight;
    }
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

Table 18.2 Member Functions of the List Classes

Function	Description
AddHead()	Adds a node to the head of the list, making the node the new head.
AddTail()	Adds a node to the tail of the list, making the node the new tail.
Find()	Searches the list sequentially to find the given object pointer. Returns a POSITION value.
FindIndex()	Scans the list sequentially, stopping at the node indicated by the given index. Returns a POSITION value for the node.
GetAt()	Gets the node at the specified position.
GetCount()	Gets the number of nodes in the list.
GetHead()	Gets the list's head node.
GetHeadPosition()	Gets the head node's position.
GetNext()	When iterating over a list, gets the next node in the list.
GetPrev()	When iterating over a list, gets the previous node in the list.
GetTail()	Gets the list's tail node.
GetTailPosition()	Gets the tail node's position.
InsertAfter()	Inserts a new node after the specified position.
InsertBefore()	Inserts a new node before the specified position.
IsEmpty()	Returns TRUE if the list is empty and returns FALSE otherwise.
RemoveAll()	Removes all of a list's nodes.
RemoveAt()	Removes a single node from a list.
RemoveHead()	Removes the list's head node.
RemoveTail()	Removes the list's tail node.
SetAt()	Sets the node at the specified position.

[an error occurred while processing this directive]

Here, the program first gets the height of the current font so that it can properly space the lines of text that it displays in the window. It then gets the number of elements in the array by calling the array object's GetSize() function. Finally, the program uses the element count to control a for loop, which calls the array object's GetAt() member function to get the value of the currently indexed array element. The program converts this value to a string for display purposes.

Removing Elements from the Array

Because it is a right button click in the window that brings up the Remove from Array dialog box, it

is the program's OnRButtonDown() function that handles the element-deletion duties. That function is shown in Listing 18.3.

### Listing 18.3 LST18\_03.cpp—The OnRButtonDown() Function

```
void CArrayView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    ArrayDlg2 dialog(this);
    dialog.m_remove = 0;
    dialog.m_removeAll = FALSE;
    int result = dialog.DoModal();
    if (result == IDOK)
    {
        if (dialog.m_removeAll)
            array.RemoveAll();
        else
            array.RemoveAt(dialog.m_remove);
        Invalidate();
    }

    CView::OnRButtonDown(nFlags, point);
}
```

In this function, after displaying the dialog box, the program checks the value of the dialog box's m\_removeAll data member. A value of TRUE means that the user has checked this option and wants to delete all elements from the array. In this case, the program calls the array object's RemoveAll() member function. Otherwise, the program calls RemoveAt(), whose single argument specifies the index of the element to delete. The call to Invalidate() forces the window to redraw the data display with the new information.

## The List Classes

**Lists** are like fancy arrays. Because lists (also called *linked lists*) use pointers to link their elements (called *nodes*) rather than depend upon contiguous memory locations to order values, lists are a better data structure to use when you need to insert and delete items quickly. However, finding items in a list can be slower than finding items in an array because a list often needs to be traversed sequentially to follow the pointers from one item to the next.

When using lists, you need to know some new vocabulary. Specifically, you need to know that the **head** of a list is the first node in the list, and the **tail** of the list is the last node in the list (see Figure 18.7). You'll see these two terms used often as you explore MFC's list classes.



**FIG. 18.7** A linked list has a head and a tail, with the remaining nodes in between.

MFC provides three list classes that you can use to create your lists. These classes are CObList (which represents a list of objects), CPtrList (which represents a list of pointers), and CStringList (which represents a list of strings). Each of these classes has similar member functions, and the classes differ in the type of data that they can hold in their lists. Table 18.2 lists and describes the

member functions of the list classes.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Introducing the List Demo Application



As you've no doubt guessed, now that you know a little about list classes and their member functions, you're going to get a chance to see lists in action. In the Chap18\list folder of this book's CD-ROM, you'll find the List Demo application. When you run the application, you see the window shown in Figure 18.8. The window displays the values of the single node with which the list begins. Each node in the list can hold two different values, both of which are integers.



**FIG. 18.8** The List Demo application starts off with one node in its list.

Using the List Demo application, you can experiment with adding and removing nodes from a list. To add a node, left-click the application's window. You then see the dialog box shown in Figure 18.9. Enter the two values that you want the new node to hold and then click OK. When you do, the program adds the new node to the tail of the list and displays the new list in the window. For example, if you were to enter the values 55 and 65 to the dialog box, you'd see the display shown in Figure 18.10.



**FIG. 18.9** A left-click in the window brings up the Add Node dialog box.



**FIG. 18.10** Each node that you add to the list can hold two different values.

You can also delete nodes from the list. To do this, right-click the window to display the Remove Node dialog box (see Figure 18.11). Using this dialog box, you can choose to remove the head or tail node. If you exit the dialog box by clicking OK, the program deletes the specified node and displays the resulting list in the window.

### Caution:

If you try to delete nodes from an empty list, the List Demo application displays a message box warning you of your error. If the application didn't catch this possible error, the program could crash when it tries to delete a nonexistent node.



**FIG. 18.11** Right-click in the window to delete a node.

## Declaring and Initializing the List

Declaring a list is as easy as declaring any other data type. Just include the name of the class that you're using, followed by the name of the object. For example, the List Demo application declares its list like this:

```
CPtrList list;
```

Here, the program is declaring an object of the `CPtrList` class. This class holds a linked list of pointers, which means that the list can reference just about any type of information.

Although there's not much that you need to do to initialize an empty list, you do need to decide what type of information will be pointed to by the pointers in the list. That is, you need to declare exactly what a node in the list will look like. The List Demo application declares a node as shown in Listing 18.4.

### Listing 18.4 LST18\_04.cpp—The `CNode` Structure

---

```
struct CNode
{
    int value1;
    int value2;
};
```

---

Here, a node is defined as a structure holding two integer values. However, you can create any type of data structure that you like for your nodes. To add a node to a list, you use the new operator to create a node structure in memory, and then you add the returned pointer to the pointer list. The List Demo application begins its list with a single node, which is created in the view class's constructor, as shown in Listing 18.5.

### Listing 18.5 LST18\_05.cpp—Creating the First Node

---

```
CNode* pNode = new CNode;
pNode->value1 = 11;
pNode->value2 = 22;
list.AddTail(pNode);
```

---

In Listing 18.5, the program first creates a new `CNode` structure on the heap and then sets the node's two members. After initializing the new node, a quick call to the list's `AddTail()` member function adds the node to the list. Because the list was empty, adding a node to the tail of the list is the same as adding the node to the head of the list. That is, the program also could have called `AddHead()` to add the node. In either case, the new single node is now both the head and tail of the list.

## Adding a Node to the List

Although you can insert nodes into a list at any position, the easiest way to add to a list is to add a

node to the head or tail, making the node the new head or tail. In the List Demo application, you left-click the window to bring up the Add Node dialog box, so you'll want to examine the `OnLButtonDown()` function, which looks like Listing 18.6.

#### **Listing 18.6 LST18\_06.cpp—List Demo's `OnLButtonDown()` Function**

---

```
void CMyListView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    // Create and initialize the dialog box.
    CAddDlg dialog;
    dialog.m_value1 = 0;
    dialog.m_value2 = 0;
    // Display the dialog box.
    int result = dialog.DoModal();
    // If the user clicked the OK button...
    if (result == IDOK)
    {
        // Create and initialize the new node.
        CNode* pNode = new CNode;
        pNode->value1 = dialog.m_value1;
        pNode->value2 = dialog.m_value2;
        // Add the node to the list.
        list.AddTail(pNode);
        // Repaint the window.
        Invalidate();
    }
    CView::OnLButtonDown(nFlags, point);
}
```

---

In Listing 18.6, after displaying the dialog box, the program checks whether the user exited the dialog box with the OK button. If so, the user wants to add a new node to the list. In this case, the program creates and initializes the new node, just as it did previously for the first node that it added in the view class's constructor. The program adds the node in the same way, too, by calling `AddTail()`. If you want to modify the List Demo application, one thing that you could try is giving the user a choice between adding the node at the head or the tail of the list, instead of just at the tail.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Deleting a Node from the List

Deleting a node from a list can be easy or complicated, depending on where in the list you want to delete the node. As with adding a node, dealing with nodes other than the head or tail requires that you first locate the node that you want and then get its position in the list. You learn about node positions in the next section, which demonstrates how to iterate over a list. To keep things simple, however, the program enables you to delete nodes only from the head or tail of the list, as shown in Listing 18.7.

### Listing 18.7 LST18\_07.cpp—The OnRButtonDown() Function

```
void CMyListView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    // Create and initialize the dialog box.
    CRemovedDlg dialog;
    dialog.m_radio = 0;
    // Display the dialog box.
    int result = dialog.DoModal();
    // If the user clicked the OK button...
    if (result == IDOK)
    {
        CNode* pNode;
        // Make sure the list isn't empty.
        if (list.IsEmpty())
            MessageBox("No nodes to delete.");
        else
        {
            // Remove the specified node.
            if (dialog.m_radio == 0)
                pNode = (CNode*)list.RemoveHead();
            else
                pNode = (CNode*)list.RemoveTail();
            // Delete the node object and repaint the window.
            delete pNode;
            Invalidate();
        }
    }
    CView::OnRButtonDown(nFlags, point);
}
```

Here, after displaying the dialog box, the program checks whether the user exited the dialog box via the OK button. If so, the program must then check whether the user wants to delete a node from the

head or tail of the list. If the Remove Head option button were checked, the dialog box's m\_radio data member would be 0. In this case, the program calls the list class's RemoveHead() member function. Otherwise, the program calls RemoveTail(). Both of these functions return a pointer to the object that was removed from the list. Before calling either of these member functions, however, notice how the program calls IsEmpty() to determine whether the list contains any nodes. You can't delete a node from an empty list!

---

**Note:**

Notice that, when removing a node from the list, the List Demo application calls delete on the pointer returned by the list. It's important to remember that, when you remove a node from a list, the node's pointer is removed from the list, but the object to which the pointer points is still in memory, where it stays until you delete it.

---

## Iterating Over the List

Often, you'll want to *iterate over* (read through) a list. You might, for example, as is the case with List Demo, want to display the values in each node of the list, starting from the head of the list and working your way to the tail. The List Demo application does exactly this in its OnDraw() function, as shown in Listing 18.8.

---

**Listing 18.8 LST18\_08.cpp—The List Demo Application's OnDraw() Function**

---

```
void CMyListView::OnDraw(CDC* pDC)
{
    CListDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    // Get the current font's height.
    TEXTMETRIC textMetric;
    pDC->GetTextMetrics(&textMetric);
    int fontHeight = textMetric.tmHeight;
    // Initialize values used in the loop.
    POSITION pos = list.GetHeadPosition();
    int displayPosition = 10;
    int index = 0;
    // Iterate over the list, displaying each node's values.
    while (pos != NULL)
    {
        CNode* pNode = (CNode*)list.GetNext(pos);
        char s[81];
        wsprintf(s, "Node %d contains %d and %d.",
            index, pNode->value1, pNode->value2);
        pDC->TextOut(10, displayPosition, s);
        displayPosition += fontHeight;
        ++index;
    }
}
```

---

In Listing 18.8, the program gets the position of the head node by calling the GetHeadPosition() member function. The position is a value that many of the list class's member functions use to quickly locate nodes in the list. You must have this starting-position value to iterate over the list.

In the while loop, the iteration actually takes place. The program calls the list object's GetNext()

member function, which requires as its single argument the position of the node to retrieve. The function returns a pointer to the node and sets the position to the next node in the list. When the position is NULL, the program has reached the end of the list. In Listing 18.8, this NULL value is the condition that's used to terminate the while loop.

## Cleaning Up the List

There's one other time when you need to iterate over a list. That's when the program is about to terminate and you need to delete all of the objects pointed to by the pointers in the list. The List Demo application performs this task in the view class's destructor, as shown in Listing 18.9.

### Listing 18.9 LST18\_09.cpp—Deleting the List's Objects

---

```
CMyListView::~CMyListView()
{
    // Iterate over the list, deleting each node.
    while (!list.IsEmpty())
    {
        CNode* pNode = (CNode*)list.RemoveHead();
        delete pNode;
    }
}
```

---

The destructor in Listing 18.9 iterates over the list in a while loop until the IsEmpty() member function returns TRUE. Inside the loop, the program removes the head node from the list (which makes the next node in the list the new head) and deletes the node from memory. When the list is empty, all the nodes that the program allocated have been deleted.

---

#### Note:

Although the List program uses RemoveHead() to remove nodes from the list, the RemoveTail() member function, which removes the last node, is a bit more efficient. In large lists, you'd probably want to use RemoveTail() to empty the list.

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 18.3 Functions of the Map Classes

Function	Description
GetCount()	Gets the number of map elements
GetNextAssoc()	When iterating over the map, gets the next element
GetStartPosition()	Gets the first element’s position
IsEmpty()	Returns TRUE if the map is empty and returns FALSE otherwise
Lookup()	Finds the value associated with a key
RemoveAll()	Removes all of the map’s elements
RemoveKey()	Removes an element from the map
SetAt()	Adds a map element or replaces an element with a matching key

Introducing the Map Demo Application

This section’s example program, Map Demo, displays the contents of a map and enables you to retrieve values from the map by giving the program the appropriate key. You can find the program in the Chap18\map folder of this book’s CD-ROM. When you run the program, you see the window shown in Figure 18.12.



**FIG. 18.12** The Map Demo application displays the contents of a map object.

The window displays the contents of the application’s map object, in which digits are used as keys to access the words that represent the numbers. To retrieve a value from the map, click in the window. You then see the dialog box shown in Figure 18.13. Type the digit that you want to use for a key and then click OK. The program finds the matching value in the map and displays it in another message box. For example, if you type 8 as the key, you see the message box shown in Figure 18.14. If the key doesn’t exist, the program’s message box tells you so.



**FIG. 18.13** The Get Map Value dialog box enables you to match a key with the key’s value in the map.



**FIG. 18.14** This message box displays the requested map value.

## Creating and Initializing the Map

The Map Demo application starts off with a ten-element map. The map object is declared as a data member of the view class, like this:

```
CMapStringToString map;
```

As you can see from the declaration, this application's map is an object of the `CMapStringToString` class, which means that the map uses strings as keys and strings as values.

Declaring the map object doesn't, of course, fill it with values. You have to do that on your own, which the Map Demo application does in its view class's constructor, as shown in Listing 18.10.

---

### Listing 18.10 LST18\_10.cpp—Initializing the Map Object

---

```
map.SetAt("1", "One");
map.SetAt("2", "Two");
map.SetAt("3", "Three");
map.SetAt("4", "Four");
map.SetAt("5", "Five");
map.SetAt("6", "Six");
map.SetAt("7", "Seven");
map.SetAt("8", "Eight");
map.SetAt("9", "Nine");
map.SetAt("10", "Ten");
```

---

The `SetAt()` function takes as parameters the key and the value to associate with the key in the map. If the key already exists, the function replaces the value associated with the key with the new value given as the second argument.

## Retrieving a Value from the Map

When you click in Map Demo's window, the Get Map Value dialog box appears, so you must suspect that the view class's `OnLButtonDown()` member function comes into play somewhere. And you'd be correct. Listing 18.11 shows this function.

---

### Listing 18.11 LST18\_11.cpp—The Map Demo Application's `OnLButtonDown()` Function

---

```
void CMapView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    // Initialize the dialog box.
    CGetDlg dialog(this);
```

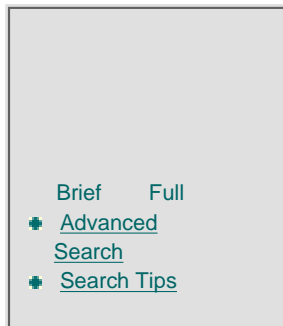
```

        dialog.m_key = "";
        // Display the dialog box.
        int result = dialog.DoModal();
// If the user exits with the OK button...
        if (result == IDOK)
        {
            // Look for the requested value.
            CString value;
            BOOL found = map.Lookup(dialog.m_key, value);
            if (found)
                MessageBox(value);
            else
                MessageBox("No matching value.");
        }
        CView::OnLButtonDown(nFlags, point);
    }

```

---

[Previous](#)
[Table of Contents](#)
[Next](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Caution:

Don't forget that you're responsible for deleting every node that you create with the `new` operator. If you fail to delete nodes, you could leave memory allocated after your program terminates. This isn't a major problem under Windows 95 because the system cleans up memory after an application exits. However, it's always good programming practice to delete any objects that you allocate in memory.

## The Map Classes

You can use MFC's *mapped collection classes* for creating lookup tables. For example, you might want to convert digits into the words that represent the numbers. That is, you might want to use the digit 1 as a key to find the word *one*. A mapped collection is perfect for this sort of task. Thanks to the many MFC map classes, you can use various types of data for keys and values.

The MFC map classes are `CMapPtrToPtr`, `CMapPtrToWord`, `CMapStringToOb`, `CMapStringToPtr`, `CMapStringToString`, `CMapWordToOb`, and `CMapWordToPtr`. The first data type in the name is the key, and the second is the value type. So, for example, `CMapStringToOb` uses strings as keys and objects as values, whereas `CMapStringToString`—which this section uses in its examples—uses strings as both keys and values. All of the map classes are similar and so have similar

member functions, which are listed and described in Table 18.3.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

In OnLButtonDown(), the program displays the dialog box in the usual way, checking to see whether the user exited the dialog box by clicking the OK button. If the user did, the program calls the map object's Lookup() member function, using the key that the user entered into the dialog box as the first argument. The second argument is a reference to the string into which the function can store the value that it retrieves from the map. If the key can't be found, the Lookup() function returns FALSE; otherwise, it returns TRUE. The program uses this return value to determine whether it should display the string value retrieved from the map or a message box indicating an error.

## Iterating Over the Map

To display the keys and values used in the map, the program must iterate over the map, moving from one entry to the next, retrieving and displaying the information for each map element. As with the array and list examples, the Map Demo application accomplishes this in its OnDraw() function, which is shown in Listing 18.12.

### Listing 18.12 LST18\_12.cpp—The Map Demo Application's OnDraw() Function

```
void CMapView::OnDraw(CDC* pDC)
{
    CMapDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    TEXTMETRIC textMetric;
    pDC->GetTextMetrics(&textMetric);
    int fontHeight = textMetric.tmHeight;
    int displayPosition = 10;
    POSITION pos = map.GetStartPosition();
    CString key;
    CString value;
    while (pos != NULL)
    {
        map.GetNextAssoc(pos, key, value);
        CString str = "Key `" + key +
            "` is associated with the value `" +
            value + "`";
        pDC->TextOut(10, displayPosition, str);
        displayPosition += fontHeight;
    }
}
```



```
    }  
}
```

---

Much of this `OnDraw()` function is similar to other versions that you've seen in this chapter. The map iteration, however, begins when the program calls the map object's `GetStartPosition()` member function, which returns a position value for the first entry in the map (not necessarily the first entry that you added to the map). Inside a while loop, the program calls the map object's `GetNextAssoc()` member function, giving the position returned from `GetStartPosition()` as the single argument. `GetNextAssoc()` retrieves the key and value at the given position and then updates the position to the next element in the map. When the position value becomes `NULL`, the program has reached the end of the map.

## Collection Class Templates

MFC includes *class templates* that you can use to create your own special types of collection classes. Although the subject of templates can be complex, using the collection class templates is easy enough. For example, suppose that you want to create an array class that can hold structures of the type shown in Listing 18.13.

### Listing 18.13 LST18\_13.cpp—A Sample Structure

---

```
struct MyValues  
{  
    int value1;  
    int value2;  
    int value3;  
};
```

---

The first step is to use the template to create your class, like this:

```
CArray<MyValues, MyValues&> myValueArray;
```

Here, `CArray` is the template that you use for creating your own array classes. The template's two arguments are the type of data to store in the array and the type of data that the new array class's member functions should use as arguments where appropriate. In this case, the type of data to store in the array is structures of the `MyValues` type. The second argument specifies that class member functions should expect references to `MyValues` structures as arguments where needed.

To build your array, you first set the array's initial size:

```
myValueArray.SetSize(10, 5);
```

Then you can start adding elements to the array, like this:

```
MyValues myValues;
```

```
myValueArray.Add(myValues) ;
```

As you can see, after you have created your array class from the template, you use the array just as you do any of MFC's array classes, as described earlier in this chapter. Other collection class templates that you can use are CList and CMap.

MFC's collection classes provide you with a way to better organize and manipulate data in your programs. Moreover, thanks to the collection class templates, you can easily create collections of any type that you need for your programs. You've probably used normal C++ arrays and maybe even linked lists in your programs, but MFC's array and list classes boot those data structures into the '90s, giving them more power than ever.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief
 Full

+ Advanced Search

+ Search Tips

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Chapter 19

# MFC Utility Classes

- Handle strings by using MFC’s CString class
- How to deal with times in an MFC program
- Handling coordinates using the CPoint class
- How to handle rectangles with the CSize and CRect classes

As you program with MFC, you run across many additional classes that, although not as immediately useful as something like the CWnd class, provide a convenient way to handle the various types of objects that are used in Windows programming. The most sophisticated of these utility classes is CString, which enables you to manipulate strings more easily. However, other classes, such as CPoint, CRect, CSize, and CTime, also do their part to make Windows programming with MFC quicker and more convenient. In this chapter, you learn about these helpful classes.

### The CString Class

There are few programs that don’t have to deal with text strings of one sort or another. Unfortunately, C++ is infamous for its difficult string-handling, whereas languages like BASIC and Pascal have always enjoyed superior power when it comes to these ubiquitous data types. MFC’s CString class addresses C++’s string problems by providing member functions that are as handy to use as those found in other languages. Table 19.1 lists the commonly used member functions of the CString class.

**Table 19.1 Commonly Used Member Functions of the CString Class**

Function	Description
Compare()	Performs a case-sensitive comparison of two strings
CompareNoCase()	Performs a case-insensitive comparison of two strings
Empty()	Clears a string
Find()	Locates a substring

GetAt()	Gets a character at a specified position in the string
GetBuffer()	Gets a pointer to the string's contents
GetLength()	Gets the number of characters in the string
IsEmpty()	Returns TRUE if the string holds no characters
Left()	Gets the left segment of a string
MakeLower()	Lowercases a string
MakeReverse()	Reverses the contents of a string
MakeUpper()	Uppercases a string
Mid()	Gets the middle segment of a string
Right()	Gets the right segment of a string
SetAt()	Sets a character at a specified position in the string
TrimLeft()	Removes leading white space characters from a string
TrimRight()	Removes trailing white space characters from a string

---

Besides the functions listed in the table, the CString class also defines a full set of operators for dealing with strings. Using these operators, you can do things like **concatenate** (join together) strings with the plus sign (+), assign values to a string object with the equal sign (=), access the string as a C-style string with the LPCTSTR operator, and more.

Creating a string object is quick and easy, like this:

```
CString str = "This is a test string";
```

Of course, there are lots of ways to construct your string object. The previous example is only one possibility. You can create an empty string object and assign characters to it later; you can create a string object from an existing string object; and you can even create a string from a repeating character. The CString class's many overloaded constructors look like this:

```
CString( )
CString( const CString& stringSrc )
CString( TCHAR ch, int nRepeat = 1 )
CString( LPCTSTR lpch, int nLength )
CString( const unsigned char* psz )
CString( LPCWSTR lpsz )
CString( LPCSTR lpsz )
```

After you have the string object created, you can call its member functions and so manipulate the string in a number of ways. For example, to convert all of the characters in the string to uppercase, you'd make a function call like this:

```
str.MakeUpper( ) ;
```

Or, to compare two strings, you'd make a function call something like this:

```
str.Compare( "Test String" );
```

You can also compare two CString objects:

```
CString testStr = "Test String";  
str.Compare(testStr);
```

---

**Note:**

If you peruse your online documentation, you'll find that most of the other CString member functions are equally as easy to use as those described here.

---

## The *CTime* and *CTimeSpan* Classes

If you've ever tried to manipulate time values returned from a computer, you'll be pleased to learn about MFC's CTime and CTimeSpan classes, which represent absolute times and elapsed times, respectively. Table 19.2 lists the member functions of the CTime class, and Table 19.3 lists the member functions of the CTimeSpan class.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief
 Full
 

+
 Advanced Search

+
 Search Tips

**BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

**Table 19.2 Member Functions of the CTime Class**

Function	Description
Format()	Constructs a string representing the time object’s time
FormatGmt()	Constructs a string representing the time object’s GMT (or UTC) time (this is the Greenwich Mean Time.)
GetCurrentTime()	Creates a CTime object for the current time
GetDay()	Gets the time object’s day as an integer
GetDayOfWeek()	Gets the time object’s day of the week, starting with 1 for Sunday
GetGmtTm()	Gets a time object’s second, minute, hour, day, month, year, day of the week, and day of the year as a tm structure
GetHour()	Gets the time object’s hour as an integer
GetLocalTm()	Gets a time object’s local time, returning the second, minute, hour, day, month, year, day of the week, and day of the year in a tm structure
GetMinute()	Gets the time object’s minutes as an integer
GetMonth()	Gets the time object’s month as an integer
GetSecond()	Gets the time object’s second as an integer
GetTime()	Gets the time object’s time as a time_t value
GetYear()	Gets the time object’s year as an integer

**Table 19.3 Member Functions of the CTimeSpan Class**

Function	Description
Format()	Constructs a string representing the time-span object’s time
GetDays()	Gets the time-span object’s days
GetHours()	Gets the time-span object’s hours for the current day
GetMinutes()	Gets the time-span object’s minutes for the current hour

GetSeconds()	Gets the time-span object's seconds for the current minute
GetTotalHours()	Gets the time-span object's total hours
GetTotalMinutes()	Gets the time-span object's total minutes
GetTotalSeconds()	Gets the time-span object's total seconds

---

## Using a *CTime* Object

Creating a *CTime* object for the current time is a simple matter of calling the `GetCurrentTime()` function, like this:

```
CTime time = CTime::GetCurrentTime();
```

Notice that, because `GetCurrentTime()` is a static member function of the *CTime* class, you can call it without actually creating a *CTime* object. You do, however, have to include the class's name as part of the function call, as shown in the preceding code. As you can see, the function returns a *CTime* object. This object represents the current time. If you wanted to display this time, you could call upon the `Format()` member function, like this:

```
CString str = time.Format("DATE: %A, %B %d, %Y");
```

The `Format()` function takes as its single argument a format string that tells the function how to create the string representing the time. The previous example creates a string that looks something like this:

```
DATE: Saturday, April 20, 1996
```

The format string used with `Format()` is not unlike the format string that you use with functions like the old DOS favorite, `printf()`, or the Windows conversion function, `wprintf()`. That is, you specify the string's format by including literal characters along with control characters. The literal characters, such as `DATE:` and the commas in the previous string example, are added to the string exactly as you type them, whereas the format codes are replaced with the appropriate values. For example, the `%A` in the previous code example will be replaced by the name of the day, and the `%B` will be replaced by the name of the month.

---

### Note:

Although the format-string concept is the same as that used with `printf()`, the `Format()` function has its own set of format codes, which are listed in Table 19.4.

---

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

• [Advanced](#)[Search](#)• [Search Tips](#)

BROWSE

BY TOPIC

**Table 19.4 Format Codes for the Format() Function**

Code	Description
%a	Day name, abbreviated (such as Sat for Saturday)
%A	Day name, no abbreviation
%b	Month name, abbreviated (such as Mar for March)
%B	Month name, no abbreviation
%c	Localized date and time (for the U.S., that would be something like 03/17/96 12:15:34)
%d	Day of the month as a number (01–31)
%H	Hour in the 24-hour format (00–23)
%I	Hour in the normal 12-hour format (01–12)
%j	Day of the year as a number (001–366)
%m	Month as a number (01–12)
%M	Minute as a number (00–59)
%p	Localized a.m./p.m. indicator for 12-hour clock
%S	Second as a number (00–59)
%U	Week of the year as a number (00–51, considering Sunday to be the first day of the week)
%w	Day of the week as a number (0–6, with Sunday being 0)
%W	Week of the year as a number (00–51, considering Monday to be the first day of the week)
%x	Localized date representation
%X	Localized time representation
%y	Year without the century prefix as a number (00–99)
%Y	Year with the century prefix as a decimal number (such as 1996)
%z	Name of time zone, abbreviated
%Z	Name of time zone, no abbreviation
%%	Percent sign

Other CTime member functions like GetMinute(), GetYear(), and GetMonth() are obvious in their usage. However, you might like an example of using a function like GetLocalTm():

```
struct tm* timeStruct;
timeStruct = time.GetLocalTm();
```

The first line of the previous code declares a pointer to a `tm` structure. (The `tm` structure is defined by Visual C++.) The second line sets the pointer to the `tm` structure created by the call to `GetLocalTm()`. Essentially, this function call retrieves all of the time information at once, organized in the `tm` structure, which is defined in the header file `TIME.H`, as shown in Listing 19.1.

#### Listing 19.1 LST19\_01.cpp—The `tm` Structure

---

```
struct tm {
    int tm_sec;      /* seconds after the minute - [0,59] */
    int tm_min;      /* minutes after the hour - [0,59] */
    int tm_hour;      /* hours since midnight - [0,23] */
    int tm_mday;      /* day of the month - [1,31] */
    int tm_mon;       /* months since January - [0,11] */
    int tm_year;      /* years since 1900 */
    int tm_wday;      /* days since Sunday - [0,6] */
    int tm_yday;      /* days since January 1 - [0,365] */
    int tm_isdst;     /* daylight savings time flag */
};
```

---

---

#### Note:

The `CTime` class features a number of overloaded constructors, enabling you to create `CTime` objects in various ways and using various times.

---

### Using a *CTimeSpan* Object

A `CTimeSpan` object is nothing more complex than the difference between two times. You can use `CTime` objects in conjunction with `CTimeSpan` objects to easily determine the amount of time that's elapsed between two absolute times. To do this, first create a `CTime` object for the current time. Then, when the time you're measuring has elapsed, create a second `CTime` object for the current time. Subtracting the old time object from the new one gives you a `CTimeSpan` object representing the amount of time that has elapsed. The example in Listing 19.2 shows how this process works.

#### Listing 19.2 LST19\_02.cpp—Calculating a Time Span

---

```
CTime startTime = CTime::GetCurrentTime();
//.
//. Time elapses...
//.
CTime endTime = CTime::GetCurrentTime();
CTimeSpan timeSpan = endTime - startTime;
```

---

### The *CPoint* Class

In many applications, you need to deal with locations of various types. For example, when a

user clicks the mouse, the system records the location of the mouse pointer. Moreover, many graphics-drawing functions use screen locations. As you know, a location on the screen is represented by an X,Y coordinate. In order to simplify handling such coordinates, MFC provides the CPoint class, which is a kind of wrapper around Windows' POINT structure.

You create a CPoint object by calling its constructor. Because the constructor is overloaded in the class, you can construct a CPoint object using various types of arguments. The many CPoint constructors look like this:

```
CPoint( );  
CPoint( int initX, int initY );  
CPoint( POINT initPt );  
CPoint( SIZE initSize );  
CPoint( DWORD dwPoint );
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The first constructor creates an uninitialized CPoint object. The other constructors create the object from two integers, from a POINT structure, from a SIZE structure, or from a double word in which the low word contains the X coordinate and the high word contains the Y coordinate.

After you've created the CPoint object, you can manipulate it using the class's member function and operators. The class has only a single member function called Offset(), which enables you to add or subtract from the point. This function is overloaded in the class, enabling you to supply various types of arguments, as follows:

```
void Offset( int xOffset, int yOffset );
void Offset( POINT point );
void Offset( SIZE size );
```

---

#### Note:

The CPoint class also features a set of operators for manipulating the object, including +, -, +=, ==, and !=.

---

The class's two data members are x and y, which, of course, hold the X and Y coordinates of the point.

## The CSize Class

In your previous Windows programming, you've probably come across SIZE structures, which hold the width and height of a rectangle. MFC encapsulates this data structure into its CSize class.

You create a CSize object by calling its constructor. Because the constructor is overloaded in the class, you can construct a CSize object using various types of arguments. The various CSize constructors look like this:

```
CSize( );
CSize( int initCX, int initCY );
CSize( SIZE initSize );
CSize( POINT initPt );
CSize( DWORD dwSize );
```

The first constructor creates an uninitialized CSize object. The other constructors create the object from two integers, from a SIZE structure, from a POINT structure, or from a double word in which the low word contains the

rectangle's width, and the high word contains the rectangle's height.

After you've created the CSize object, you can manipulate it using the class's operators, which include +, -, +=, -=, ==, and !=. The class's two data members are cx and cy, which hold the width and height of the rectangle.

## The CRect Class

Rectangles are everywhere in Windows programming. Although the CSize class enables you to specify the width and height of a rectangle, the class doesn't hold a rectangle's position. To fully specify a rectangle—including its position and size—you need an object of MFC's CRect class, which encapsulates Windows' RECT structure.

You create a CRect object by calling its constructor. Because, like the CPoint and CSize classes, the constructor is overloaded in the class, you can construct a CRect object, using various types of arguments. The CRect constructors look like this:

```
CRect( );  
CRect( int l, int t, int r, int b );  
CRect( const RECT& srcRect );  
CRect( LPCRECT lpSrcRect );  
CRect( POINT point, SIZE size );  
CRect( POINT topLeft, POINT bottomRight );
```

The first constructor creates an uninitialized CRect object. The other constructors create the object from the following:

- Four integers (representing the rectangle's left, top, right, and bottom coordinates)
- A RECT structure
- A pointer to a RECT structure
- A POINT and SIZE structure (where the POINT structure holds the rectangle's position, and the SIZE structure holds the rectangle's width and height)
- Two POINT structures (one representing the rectangle's position and the other representing the rectangle's size)

After creating the CRect object, you can manipulate it using the class's member functions and operators. The class's member functions are listed in Table 19.6 along with their descriptions.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 20.1 Thread Priority Constants

Constant	Description
THREAD_PRIORITY_ABOVE_NORMAL	Sets a priority one point higher than normal.
THREAD_PRIORITY_BELOW_NORMAL	Sets a priority one point lower than normal.
THREAD_PRIORITY_HIGHEST	Sets a priority two points above normal.
THREAD_PRIORITY_IDLE	Sets a base priority of 1. For a REALTIME_PRIORITY_CLASS process, sets a priority of 16.
THREAD_PRIORITY_LOWEST	Sets a priority two points below normal.
THREAD_PRIORITY_NORMAL	Sets normal priority.
THREAD_PRIORITY_TIME_CRITICAL	Sets a base priority of 15. For a REALTIME_PRIORITY_CLASS process, sets a priority of 31.

Note:

A thread’s priority determines how often the thread takes control of the system relative to the other running threads. Generally, the higher the priority, the more running time the thread gets, which is why the value of THREAD\_PRIORITY\_TIME\_CRITICAL is so high.

To see a simple thread in action, build the Thread application as detailed in the following steps.

The complete source code and executable file for this part of the Thread application can be found in the CHAP20\Thread, Part 1 directory on this book’s CD-ROM.

1. Start a new AppWizard project workspace called Thread, as shown in Figure 20.1.



**FIG. 20.1** Here’s how to start the Thread project.

2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 20.2.
 

Step 1: Single document  
 Step 2: Default settings  
 Step 3: Default settings  
 Step 4: Turn off all options  
 Step 5: Default settings  
 Step 6: Default settings



**FIG. 20.2** These are the AppWizard settings for the Thread project.

3. Use the resource editor to add a Thread menu to the application's menu bar. Give the menu one command called Start Thread with a command ID of ID\_STARTTHREAD, as shown in Figure 20.3.



**FIG. 20.3** Add a Thread menu with a Start Thread command.

4. Use ClassWizard to associate the ID\_STARTTHREAD command with the OnStartthread() message response function, as shown in Figure 20.4. Make sure that you have CThreadView selected in the Class Name box before you add the function.



**FIG. 20.4** Add the OnStartthread() message response function to the view class.

5. Click the Edit Code button and then add the following lines to the new OnStartthread() function, right after the TODO: Add your command handler code here comment:

```

        HWND hWnd = GetSafeHwnd();
        AfxBeginThread(ThreadProc, hWnd, THREAD_PRIORITY_NORMAL);

```

6. Add the function shown in Listing 20.1 to the program, being sure to place it right before the OnStartthread() function. Note that ThreadProc() is a global function and not a member function of the CThreadView class, even though it's placed in the view class's implementation file.

#### **Listing 20.1 LST20\_01.cpp—The ThreadProc() Function**

---

```

UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);

    return 0;
}

```

---

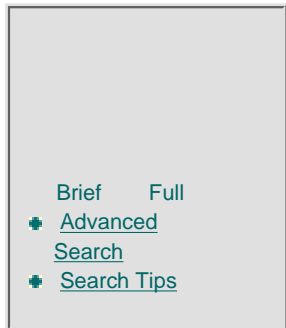
You've now completed the first version of the Thread application. When you run the Thread program, the main window appears. Select the Thread, Start Thread command, and the system starts the thread represented by the ThreadProc() function and displays a message box, as shown in Figure 20.5.





**FIG. 20.5** The simple secondary thread in the Thread program displays a message box and then ends.

[Previous](#) [Table of Contents](#) [Next](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 20 Programming Threads

- Learning how to create and run threads
- Discovering inter-thread communication
- Synchronizing threads with critical sections
- Using mutexes to synchronize threads
- Using semaphores to manage multiple resource accesses

You know that when you are using Windows 95 (and other modern operating systems), you can run several programs simultaneously. This ability is called **multitasking**. What you might not know is that many of today's operating systems also allow **threads**, which are processes that are kind of a step down from a complete application. A thread is a lot like a subprogram. An application can create several threads—several different flows of execution—and run them concurrently. Threads give you the ability to have multitasking inside of multitasking. The user knows that he or she can run several applications at a time. The programmer knows that each application can run several threads at a time. In this chapter, you learn how to create and manage threads in your applications.

### Understanding Simple Threads

When you come right down to it, a thread is little more than a function that the system runs concurrently with the main program. That is, to create a thread using MFC, all you have to do is write a function that represents the thread. Then, call `AfxBeginThread()` to start the thread. The thread remains active as long as the thread's function is executing. When the thread function exits, the thread is destroyed. A simple call to `AfxBeginThread()` looks like this:

```
AfxBeginThread(ProcName, param, priority);
```

In the previous line, ProcName is the name of the thread's function; param is any 32-bit value that you want to pass to the thread; and priority is the thread's priority, which is represented by a number of predefined constants. Those constants and their descriptions are shown in Table 20.1.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 ● [Advanced Search](#)  
 ● [Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

If you examine the Thread program's source code, you see that when you select the Thread, Start Thread command, MFC calls the OnStarthread() function, which starts the secondary thread with this line:

```
AfxBeginThread(ThreadProc, hWnd, THREAD_PRIORITY_NORMAL);
```

The function's first argument indicates that the thread is represented by the ThreadProc() function. That is, to start the thread, the system calls ThreadProc() and runs ThreadProc() concurrently with the program's main thread, almost as if ThreadProc() were a completely separate program. The second argument in the preceding call to AfxBeginThread() is the application's main window handle (although it can be any 32-bit value that you want to pass to the thread), and the third argument is the requested thread priority. Because the ThreadProc() thread has no critical processor time-sharing needs, the priority is set to a normal setting.

---

**Note:**

Any application always has at least one thread, which is the program's primary or main thread. You can start and stop as many additional threads as you need, but the main thread keeps running as long as the application is active.

---

## Understanding Thread Communication

Usually, a secondary thread performs some sort of task for the main program, which implies that there needs to be a channel of communication between the program (which is also a thread) and its secondary threads. There are several ways to accomplish these communication tasks: using global variables, using event objects, and using messages. In this section, you'll explore these thread-communication techniques.

### Communicating with Global Variables

Suppose you want your main program to be able to stop the thread. You need a way, then, to tell the thread when to stop. One way to do this would be to set up a global variable and then have the thread monitor the global variable for a value that signals the thread to end. To see how this technique works, perform the following steps to modify the Thread application.

The complete source code and executable file for this part of the Thread application can be found in the CHAP20\Thread, Part 2 directory on this book's CD-ROM.



1. Use the resource editor to add a Stop Thread command to the application's Thread menu. Give this new command the ID\_STOPTHREAD ID, as shown in Figure 20.6.



**FIG. 20.6** Add a Stop Thread command to the Thread menu.

2. Use ClassWizard to associate the ID\_STOPTHREAD command with the OnStopthread() message response function, as shown in Figure 20.7. Make sure that you have CThreadView selected in the Class Name box before you add the function.



**FIG. 20.7** Add the OnStopthread() message response function.

3. Add the following line to the OnStopthread() function, right after the TODO: Add your command handler code here comment:

```
threadController = 0;
```

4. Add the following line to the top of the ThreadView.cpp file, right after the endif directive:

```
volatile int threadController;
```

5. Add the following line to the OnStartthread() function, right after the TODO: Add your command handler code here comment:

```
threadController = 1;
```

6. Replace the ThreadProc() function with the one shown in Listing 20.2.

---

**Listing 20.2 LST20\_02.cpp—The New ThreadProc() Function**

---

```
UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);

    while (threadController == 1);

    ::MessageBox((HWND)param, "Thread stopped.", "Thread", MB_OK);

    return 0;
}
```

---

You've now completed the second version of the Thread application. When you run the program, select the Thread, Start Thread command to start the secondary thread. When you do, a message box appears telling you that the new thread was started. To stop the thread, select the Thread, Stop Thread command. Again, a message box appears, this time telling you that the thread is stopping.

The main program communicates with the thread through the global variable threadController. Before starting the thread, the main program sets this variable to 1, which indicates to the thread that it should continue to run. After setting the variable, the main program starts the thread, which is represented by the ThreadProc() function.

In ThreadProc(), the thread first displays a message box, telling the user that the thread is starting. Then a while loop continues to poll the threadController global variable, waiting for its value to change to 0. Although this while loop is trivial, it is here that you would place the code that performs whatever task you want the thread to perform, being sure not to tie things up for too long before rechecking the value of threadController.

When the user selects the Thread, Stop Thread command, MFC calls the OnStopthread() function, where the main program sets threadController to 0. This action causes the thread function to exit its while loop, after which it displays the final message box and exits.

---

**Caution:**

Using global variables to communicate between threads is, to say the least, an unsophisticated approach to thread communication and can be a dangerous technique if you're not sure how C++ handles variables from an assembly-language level. Other thread-communication techniques are safer and more elegant.

---

## Communicating with User-Defined Messages

Now you have a simple, albeit unsophisticated, method for communicating information from your main program to your thread. How about the reverse? That is, how can your thread communicate with the main program? The easiest method to accomplish this communication is to incorporate user-defined Windows messages into the program.

The first step is to define a user message, which you can do easily, like this:

```
const WM_USERMSG = WM_USER + 100;
```

The WM\_USER constant, defined by Windows, holds the first available user-message number. Because MFC might use some of the user messages for its own purposes, the preceding line sets WM\_USERMSG to WM\_USER+100.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

After defining the message, you call `::PostMessage()` from the thread to send the message to the main program whenever you need to. Such a code line might look like this:

```
::PostMessage((HWND)param, WM_USERMSG, 0, 0);
```

`PostMessage()`'s four arguments are the handle of the window to which the message should be sent, the message identifier, and the message's `WPARAM` and `LPARAM` parameters.

#### Tip:

The double colons in front of a function name indicate a call to a Windows API function, rather than an MFC class member function. You can use the colons to force the program to call an original Windows function rather than the one defined in an MFC class. For example, inside an MFC window class, you can call `MessageBox("Hi, There!")` to display "Hi, There!" to the user. This form of `MessageBox()` is a member function of the MFC window classes. To call the original Windows version, you'd write something like `::MessageBox(0, "Hi, There!", "Message", MB_OK)`. Notice the colons in front of the function name (not to mention the additional function arguments).

Modify the Thread application according to the next steps to see how to post user messages from a thread:



The complete source code and executable file for this part of the Thread application can be found in the CHAP20\Thread, Part 3 directory on this book's CD-ROM.

1. Add the following line to the top of the ThreadView.h header file, right before the beginning of the class declaration:

```
const WM_THREADENDED = WM_USER + 100;
```

2. Still in the header file, add the following line to the class's Implementation section, right after the protected keyword:

```
afx_msg LONG OnThreadended(WPARAM wParam, LPARAM lParam);
```

3. Load the ThreadView.cpp file, and then add the following line to the class's message map, making sure to place it right *after* the `}}AFX_MSG_MAP` comment:

```
ON_MESSAGE(WM_THREADENDED, OnThreadended)
```

4. Replace the ThreadProc() function with the one shown in Listing 20.3.

#### Listing 20.3 LST20\_03.cpp—The New ThreadProc() Function

```

UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);

    while (threadController == 1);

    ::PostMessage((HWND)param, WM_THREADENDED, 0, 0);

    return 0;
}
    
```

---

5. Add the function shown in Listing 20.4 to the end of the ThreadView.cpp file.

---

**Listing 20.4 LST20\_04.cpp—The OnThreadended() Function**

---

```
LONG CThreadView::OnThreadended(WPARAM wParam, LPARAM lParam)
{
    MessageBox( "Thread ended." );
    return 0;
}
```

---

You've now completed the third version of the Thread application. When you run the new version, select the Thread, Start Thread command to start the thread. When you do, a message box appears telling you that the thread has started. To end the thread, select the Thread, Stop Thread command. Just as with the previous version of the program, a message box appears, telling you that the thread has ended.

Although this version of the Thread application seems to run identically to the previous version, there's a subtle difference. Now the program displays the message box that signals the end of the thread in the main program rather than from inside the thread. The program can do this because, when the user selects the Stop Thread command, the thread sends a WM\_THREADENDED message to the main program. When the program receives that message, it displays the final message box.

## Communicating with Event Objects

A slightly more sophisticated method of signaling between threads is to use *event objects*, which under MFC are represented by the CEvent class. An event object can be in one of two states: signaled and nonsignaled. Threads can watch for events to be signaled and so perform their operations at the appropriate time. Creating an event object is as easy as declaring a global variable, like this:

```
CEvent threadStart;
```

Although the CEvent constructor has a number of optional arguments, you can usually get away with creating the default object, as shown in the previous line of code. Upon creation, the event object is automatically in its nonsignaled state. To signal the event, you call the event object's SetEvent() member function, like this:

```
threadStart.SetEvent();
```

After the preceding line executes, the threadStart event object will be in its signaled state. Your thread should be watching for this signal so that the thread knows it's okay to get to work. How does a thread watch for a signal? By calling the Windows API function, WaitForSingleObject():

```
::WaitForSingleObject(threadStart.m_hObject, INFINITE);
```

This function's two arguments are the handle of the event for which to check and the length of time for which the function should wait for the event. (The event's handle is stored in the event object's m\_hObject data member.) The predefined INFINITE constant tells WaitForSingleObject() not to return until the specified event is signaled. In other words, if you place the preceding line at the beginning of your thread, the system suspends the thread until the event is signaled. Even though you've started the thread execution, it's halted until whatever you need to have happen happens. When your program is ready for the thread to perform its duty, you call the SetEvent() function, as described a couple of paragraphs ago.

As soon as the thread is no longer suspended, it can go about its business. But if you want to signal the end of the thread from the main program, the thread must watch for this next event to be signaled. The thread can do this by polling for the event. To poll for the event, you again call WaitForSingleObject(); only this time you give the function a wait time of 0, like this:

```
::WaitForSingleObject(threadend.m_hObject, 0);
```

In this case, if WaitForSingleObject() returns WAIT\_OBJECT\_0, the event has been signaled. Otherwise, the event still is in its nonsignaled state.

To better see how event objects work, follow the steps given next to further modify the Thread application:



The complete source code and executable file for this part of the Thread application can be found in the CHAP20\Thread, Part 4 directory on this book's CD-ROM.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

+ [Advanced](#)[Search](#)+ [Search Tips](#)BROWSE  
BY TOPIC

1. Add the following line to the top of the ThreadView.cpp file, right after the line

```
#include "ThreadView.h":
#include "afxmt.h"
```

2. Add the following lines near the top of the ThreadView.cpp file, after the volatile int threadController line that you placed there previously:

```
CEvent threadStart;
CEvent threadEnd;
```

3. Delete the volatile int threadController line from the file.
4. Replace the ThreadProc() function with the one shown in Listing 20.5.

#### Listing 20.5 LST20\_05.cpp—Yet Another New ThreadProc() Function

```
UINT ThreadProc(LPVOID param)
{
    ::WaitForSingleObject(threadStart.m_hObject, INFINITE);
    ::MessageBox((HWND)param, "Thread activated.",
        "Thread", MB_OK);

    BOOL keepRunning = TRUE;
    while (keepRunning)
    {
        int result =
            ::WaitForSingleObject(threadEnd.m_hObject, 0);
        if (result == WAIT_OBJECT_0)
            keepRunning = FALSE;
    }

    ::PostMessage((HWND)param, WM_THREADENDED, 0, 0);

    return 0;
}
```

5. Replace the code in the OnStartthread() function with the following line:

```
threadStart.SetEvent();
```

6. Replace the code in the OnStopthread() function with the following line:

```
threadEnd.SetEvent();
```

7. Use ClassWizard to add an OnCreate() function, as shown in Figure 20.8. Make sure that you have CThreadView selected in the Class Name box before you add the function.



**FIG. 20.8** Here's ClassWizard when you add the OnCreate() function.

8. Add the following lines to the OnCreate() function, right after the TODO: Add your specialized creation code here comment:

```
HWND hWnd = GetSafeHwnd();  
AfxBeginThread(ThreadProc, hWnd);
```

You've now completed the next version of the Thread application. Again, this new version of the program seems to run the same way as the previous version. However, the program is now using both event objects and user-defined Windows messages to communicate between the main program and the thread. No more messing with clunky global variables.

One big difference from previous versions of the program is that the secondary thread gets started in the OnCreate() function, which is called when the application first starts up. However, because the first line of the thread function is the call to WaitForSingleObject(), the thread immediately suspends execution and waits for the threadStart event to be signaled.

When the threadStart event object is signaled, the thread is free to display the message box and then enter its while loop, where it polls the threadEnd event object. The while loop continues to execute until threadEnd becomes signaled, at which time the thread sends the WM\_THREADENDED message to the main program and exits. Because the thread is started in OnCreate(), after the thread ends, it cannot be restarted.

## Using Thread Synchronization

Using multiple threads can lead to some interesting problems. For example, how do you prevent two threads from accessing the same data at the same time? What if, for example, one thread is in the middle of trying to update a data set while another thread tries to read that data? The second thread will almost certainly read corrupted data, because only some of the data set will have been updated.

Keeping threads working together properly is called **thread synchronization**. Event objects, about which you just learned, are actually a form of thread synchronization. In this section, you'll learn about **critical sections**, **mutexes**, and **semaphores**—thread synchronization objects that make your thread programming even safer.

### Using Critical Sections

Critical sections are an easy way to ensure that only one thread at a time can access a data set. When you use a critical section, you give your threads an object that they have to share between them. Whichever thread possesses the critical-section object has access to the guarded data. Other threads have to wait until the first thread releases the critical section object, after which time another thread can grab the critical section object to access the data in turn.

Because the guarded data is represented by a single critical-section object and because only

one thread can own the critical section object at any given time, the guarded data can never be accessed by more than a single thread at a time.

To create a critical-section object in an MFC program, you create an instance of the `CCriticalSection` class, like this:

```
CCriticalSection criticalSection;
```

Then, when program code is about to access the data that you want to protect, you call the critical-section object's `Lock()` member function, like this:

```
criticalSection.Lock();
```

If another thread doesn't already own the critical section object, `Lock()` gives the object to the calling thread. That thread can then access the guarded data, after which it calls the critical-section object's `Unlock()` member function:

```
criticalSection.Unlock();
```

`Unlock()` releases the ownership of the critical-section object so that another thread can grab it and access the guarded data.

The best way to implement something like critical sections is to build the data that you want to protect into a thread-safe class. When you do this, you no longer have to worry about thread synchronization in the main program; the class handles it all for you. As an example, look at Listing 20.6, which is the header file for a thread-safe array class.

#### **Listing 20.6 CountArray.h—The CCountArray Class's Header File**

---

```
#include "afxmt.h"

class CCountArray
{
private:
    int array[10];
    CCriticalSection criticalSection;

public:
    CCountArray() {};
    ~CCountArray() {};

    void SetArray(int value);
    void GetArray(int dstArray[10]);
};
```

---

The header file starts off by including the MFC header file, `afxmt.h`, which gives the program access to the `CCriticalSection` class. Within the `CCountArray` class declaration, the file declares a ten-element integer array, which is the data that the critical section will guard, and declares the critical-section object, here called `criticalSection`. The `CCountArray` class's public member functions include the usual constructor and destructor, as well as functions for setting and reading the array. These latter two member functions must deal with the critical-section object, because these functions access the array.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The second thread, named `ReadThreadProc()`, is also trying to access the same critical-section object to construct a display string of the values contained in the array. But if `WriteThreadProc()` is currently trying to fill the array with new values, `ReadThreadProc()` has to wait. The inverse is also true. That is, `WriteThreadProc()` can't access the guarded data until it can regain ownership of the critical section from `ReadThreadProc()`.

If you really want to prove that the critical-section object is working, remove the `criticalSection.Unlock()` line from the end of the `CCountArray` class's `SetArray()` member function. Then compile and run the program. This time when you start the threads, no message box appears until the array has been set to all 9s. Why? Because `WriteThreadProc()` takes the critical-section object and doesn't let it go until the thread ends, which forces the system to suspend `ReadThreadProc()` until the first thread executes its return instruction.

## Using Mutexes

Mutexes (the word "mutex" is formed from "mutually exclusive") are a lot like critical section objects but are a little more complicated, because they enable safe sharing of resources, not only between threads in the same application, but also between threads of different applications. Although synchronizing threads of different applications is beyond the scope of this book, you can get a little experience with mutexes by using them in place of critical sections.

Listing 20.9 is the `CCountArray2` class's header file. Except for the new class name and the mutex object, this header file is identical to the original `CountArray.h`. Listing 20.10 is the modified class's implementation file. As you can see, the member functions look a lot different when they are using mutexes instead of critical sections, even though both objects provide essentially the same type of services.

### Listing 20.9 CountArray.h—The CCountArray2 Class's Header File

```
#include "afxmt.h"

class CCountArray2
{
private:
    int array[10];
    CMutex mutex;

public:
```

```

    CCountArray2() {} ;
    ~CCountArray2() {} ;

    void SetArray(int value);
    void GetArray(int dstArray[10]);
};

```

---

### **Listing 20.10 CountArray2.cpp—The CCountArray2 Class’s Implementation File**

---

```

#include "stdafx.h"
#include "CountArray2.h"

void CCountArray2::SetArray(int value)
{
    CSingleLock singleLock(&mutex);
    singleLock.Lock();

    for (int x=0; x<10; ++x)
        array[x] = value;
}

void CCountArray2::GetArray(int dstArray[10])
{
    CSingleLock singleLock(&mutex);
    singleLock.Lock();

    for (int x=0; x<10; ++x)
        dstArray[x] = array[x];
}

```

---

In order to access a mutex object, you must create a `CSingleLock` or `CMultiLock` object, which performs the actual access control. The `CCountArray2` class uses `CSingleLock` objects, because this class is dealing with only a single mutex. When the code is about to manipulate guarded resources (in this case, the array), you create a `CSingleLock` object, like this:

```
CSingleLock singleLock(&mutex);
```

The constructor’s argument is a pointer to the thread-synchronization object that you want to control. Then, to gain access to the mutex, you call the `CSingleLock` object’s `Lock()` member function:

```
singleLock.Lock();
```

If the mutex is unowned, the calling thread becomes the owner. If another thread already owns the mutex, the system suspends the calling thread until the mutex is released, at which time the waiting thread wakes up and takes control

of the mutex.

To release the mutex, you call the `CSingleLock` object's `Unlock()` member function. However, if you create your `CSingleLock` object on the stack (rather than on the heap, using the `new` operator) as shown in Listing 20.10, you don't have to call `Unlock()` at all. When the function exits, the object goes out of scope, which causes its destructor to execute. The destructor automatically unlocks the object for you.



The complete source code and executable file for this part of the Thread application can be found in the `CHAP20\Thread, Part 6` directory on this book's CD-ROM.

To try out the new `CCountArray2` class in the Thread application, place the `CountArray2.h` and `CountArray2.cpp` files in the Thread project's folder, and then delete the original `CountArray.h` and `CountArray.cpp` files from the project. Finally, add the `CountArray2.cpp` file to the project and, in `ThreadView.cpp`, change all references to `CCountArray` to `CCountArray2`. (Be sure to change the `#include "CountArray.h"` line to `#include "CountArray2.h"`.) Because all the thread synchronization is handled in the `CCountArray2` class, no further changes are necessary to use mutexes rather than critical sections. Convenient, eh?

## Using Semaphores

Although semaphores are used like critical sections and mutexes in an MFC program, they serve a slightly different function. Rather than letting only one thread access a resource, semaphores let multiple threads access a resource, but only to a point. That is, semaphores let a maximum number of threads access a resource simultaneously.

When you create the semaphore, you tell it how many threads should be allowed simultaneous access to the resource. Then, each time a thread grabs the resource, the semaphore decrements its internal counter. When the counter reaches 0, no further threads are allowed access to the guarded resource until another thread releases the resource, which increments the semaphore's counter.

You create a semaphore by supplying the initial count and the maximum count, like this:

```
CSemaphore Semaphore(2, 2);
```

Because, in this section, you'll be using a semaphore to create a thread-safe class, it's actually more convenient to declare a `CSemaphore` pointer as a data member of the class and then create the `CSemaphore` object dynamically in the class's constructor, like this:

```
semaphore = new CSemaphore(2, 2);
```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 21

# Exceptions and Other Power Features

- Discover how to catch, throw, and define exception objects
- Learn about function templates
- Find out how to create and use class templates
- Learn about Run-Time Type Information
- Learn how to define and use namespaces

C++ is an evolving language, and, as such, it frequently undergoes review and improvement. New power features that have been added to C++ and MFC in the recent past are exceptions, templates, Run-Time Type Information (RTTI), and namespace support. These are programming issues that will become more and more important to Windows developers. For that reason, learning to implement these new features into your programming projects is a must-do task.

## Compiling and Running Console Applications

All of the programs in this chapter are console applications, which means that they are DOS-style programs that don't take advantage of the Windows GUI. The programs were written as console applications to avoid obscuring the important programming details with tons of Windows-specific program code. To compile and run a console application, complete the following steps.

1. Choose File, New from Developer Studio's menu bar. The New property sheet appears.
2. Choose Win32 Console Application in the property sheet's left pane, type the project's name into the Project Name box, and set the Location box to the directory in which you want the new project to be created.
3. Click OK to finalize your settings. Developer Studio creates the new project.
4. Copy the program's source code into the project's directory.
5. Choose Project, Add to Project, Files from Developer Studio's menu bar. The Insert Files Into Project dialog box appears.
6. Double-click the source code file to add it to the project.
7. Click the Build button, or choose Build, Build from the menu bar. Developer Studio compiles and links the program.

8. Run the program's EXE file from a DOS window. (You can start a DOS window from Windows 95's Start, Programs menu.)

## Understanding Exceptions

When you write applications using Visual C++ and MFC, sooner or later you're going to run into *exceptions*. An exception is a special type of error object that is created when something goes wrong in a program. After Visual C++ creates the exception object, it sends it to your program, an action called *throwing an exception*. It's up to your program to *catch* the exception. You enable your program to do this by writing the exception-handling code. In this section, you get the inside info on these important error-handling objects.

### Simple Exception Handling

The mechanism used by exception-handling code is really pretty simple. You place the source code that you want guarded against errors inside a try block. You then construct a catch program block that acts as the error handler. If the code in the try block (or any code called from the try block) generates an exception (called *throwing an exception*, remember), the try block immediately ceases execution, and the program continues inside the catch block.

For example, memory allocation is one place in a program where you might expect to run into trouble. Listing 21.1 shows a nonsensical little program that allocates some memory and then immediately deletes it. Because memory allocation could fail, the code that allocates the memory is enclosed in a try program block. If the pointer returned from the memory allocation is NULL, the try block throws an exception. In this case, the exception object is a string.

---

#### Note:

The complete source code and executable file for the Exception application can be found in the CHAP21\Exception directory of this book's CD-ROM.

---



---

#### Listing 21.1 Exception.cpp—Simple Exception Handling

---

```
#include <iostream.h>

int main()
{
    int* buffer;
    char* msg[] = {"Memory allocation failed!"};

    try
    {
        buffer = new int[256];

        if (buffer == NULL)
            throw *msg;
    }
}
```

```
        else
            delete buffer;
    }
    catch(char* exception)
    {
        cout << exception << endl;
    }

    return 0;
}
```

---

When the program throws the exception, program execution jumps to the first line of the catch program block. In the case of Listing 21.1, this line just prints out a message, after which the function's return line is executed and the program ends.

If the memory allocation is successful, the program executes the entire try block, deleting the buffer. Then program execution skips over the catch block completely, in this case going directly to the return statement.

---

**Note:**

The `catch` program block does more than direct program execution. It actually catches the `exception` object thrown by the program. For example, in Listing 21.1, you can see the exception object being caught inside the parentheses following the `catch` keyword. This is very similar to a parameter being received by a method. In this case, the type of "parameter" is `char*`, and the name of the parameter is `exception`.

---

## ***Exception Objects***

The beauty of C++ exceptions is that the exception object thrown can be just about any kind of data structure that you like. For example, you might want to create an exception class for certain kinds of exceptions that occur in your programs. Listing 21.2 shows a program that defines a general-purpose exception class called `CMyException`. In the case of a memory allocation failure, the main program creates an object of the class and throws it. The catch block catches the `CMyException` object, calls the object's `GetError()` member function to get the object's error string, and then displays the string on the screen.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

➤ [Advanced Search](#)

➤ [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)**Note:**

The complete source code and executable file for the Exception2 application can be found in the CHAP21\Exception2 directory of this book's CD-ROM.

**Listing 21.2 Exception2.cpp—Creating an Exception Class**

```
#include <iostream.h>

class CMyException
{
protected:
    char* m_msg;

public:
    CMyException(char *msg)
    {
        m_msg = msg;
    };

    ~CMyException(){};

    char* GetError()
    {
        return m_msg;
    };
};

int main()
{
    int* buffer;

    try
    {
        buffer = new int[256];

        if (buffer == NULL)
        {
            CMyException* exception =
                new CMyException("Memory allocation failed!");
            throw exception;
        }
    }
}
```

```

        }
        else
            delete buffer;
    }
    catch(CMyException* exception)
    {
        char* msg = exception->GetError();
        cout << msg << endl;
    }

    return 0;
}

```

---

An exception object can be as simple as an integer error code or as complex as a fully developed class. Whatever works best for your program is the way to go. You might, for example, want to derive specific types of exception classes from the general exception class developed in Listing 21.2.

### Placing the *catch* Block

The catch program block doesn't have to be in the same function as the one in which the exception is thrown. When an exception is thrown, the system starts "unwinding the stack," looking for the nearest catch block. If the catch block is not found in the function that threw the exception, the system looks in the function that called the throwing function. This search continues on up the function call stack. If the exception is never caught, the program halts.

Listing 21.3 is a short program that demonstrates this concept. The program throws the exception from the AllocateBuffer() function but catches the exception in main(), which is the function from which AllocateBuffer() is called.

---

#### Note:

The complete source code and executable file for the Exception3 application can be found in the CHAP21\Exception3 directory of this book's CD-ROM.

---

### Listing 21.3 Exception3.cpp—Catching Exceptions Outside of the Throwing Function

---



```

#include <iostream.h>

class CMyException
{
protected:
    char* m_msg;

public:
    CMyException(char *msg)
    {
        m_msg = msg;
    };
};

```

```

~CMyException(){};

char* GetError()
{
    return m_msg;
};

};

int* AllocateBuffer()
{
    int* buffer = new int[256];

    if (buffer == NULL)
    {
        CMyException* exception =
            new CMyException("Memory allocation failed!");
        throw exception;
    }

    return buffer;
}

int main()
{
    int* buffer;

    try
    {
        buffer = AllocateBuffer();
        delete buffer;
    }
    catch(CMyException* exception)
    {
        char* msg = exception->GetError();
        cout << msg << endl;
    }

    return 0;
}

```

---

## Handling Multiple Types of Exceptions

Because it's often the case that a block of code generates more than one type of exception, you can use multiple catch blocks with a try block. You might, for example, need to be on the lookout for both CMyException and char\* exceptions. Because a catch block must receive a specific type of exception object (except in a special case that you'll learn about next), you need two different catch blocks to watch for both CMyException and char\* exception objects.

The special case I referred to in the previous paragraph is a catch block that can receive any type of exception object. You define this type of catch block by placing ellipses in the parentheses, rather than a specific argument. The problem with this sort of multipurpose

catch block is that you have no access to the exception object received and so must handle the exception in some general way.

Listing 21.4 is a program that generates three different types of exceptions based on a user's input. When you run the program, you're instructed to enter a value between 4 and 8, except for 6. If you enter a value less than 4, the program throws a CMyException exception; if you enter a value greater than 8, the program throws a char\* exception. If you happen to enter 6, the program throws the entered value as an exception.

Although the program throws the exceptions in the GetValue() function, the program catches them all in main(). The try block in main() is associated with three catch blocks. The first catches the CMyException object; the second catches the char\* object; and the third catches any other exception that happens to come down the pike.

---

**Caution:**

Just as with if-else statements, the order in which you place catch program blocks can have a profound effect on program execution. You should always place the most specific catch blocks first. For example, in Listing 21.4, if the catch(...) block was first, none of the other catch blocks would ever be called. This is because the catch(...) is as general as you can get, catching every single exception that the program throws. In this case (as in most cases), you want to use catch(...) to receive only the leftover exceptions.

---

**Note:**

The complete source code and executable file for the Exception4 application can be found in the CHAP21\Exception4 directory of this book's CD-ROM.

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

• [Advanced](#)[Search](#)• [Search Tips](#)BROWSE  
BY TOPIC

## Listing 21.4 Exception4.cpp—Using Multiple Catch Blocks



```

#include <iostream.h>

class CMyException
{
protected:
    char* m_msg;

public:
    CMyException(char *msg)
    {
        m_msg = msg;
    };

    ~CMyException(){};

    char* GetError()
    {
        return m_msg;
    };
};

int GetValue()
{
    int value;

    cout << "Type a number from 4 to 8 (except 6):" << endl;
    cin >> value;

    if (value < 4)
    {
        CMyException* exception =
            new CMyException("Value less than 4!");
        throw exception;
    }
    else if (value > 8)
    {
        throw "Value greater than 8!";
    }
}

```



```

    }
    else if (value == 6)
    {
        throw value;
    }
    return value;
}

int main()
{
    try
    {
        int value = GetValue();
        cout << "The value you entered is okay." << endl;
    }
    catch(CMyException* exception)
    {
        char* msg = exception->GetError();
        cout << msg << endl;
    }
    catch(char* msg)
    {
        cout << msg << endl;
    }
    catch(...)
    {
        cout << "Caught unknown exception!" << endl;
    }

    return 0;
}

```

---



---

**Note:**

When your program catches an exception, it has several courses of action at its disposal. It can handle the exception; it can rethrow the exception for handling somewhere else in the program; or it can both handle the exception and rethrow it. To rethrow an exception, use the `throw` keyword with no argument.

---



---

**Note:**

In older C++ program code, it is 1

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)

 **BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

---

**Note:**

Notice how, in Listing 21.6, the `Min()` template uses the data type `Type`, not only in its parameter list and function argument list, but also in the body of the function, to declare a local variable. This illustrates how you can use the parameter types just as you would use any specific data type such as `int` or `char`.

---

Because function templates are so flexible, they can often lead to trouble. For example, in the `Min()` template, you have to be sure that the data types that you supply as parameters can be compared. If you tried to compare two classes, your program would not compile unless the classes overloaded the `<` and `>` operators.

Another way you can run into trouble is when the arguments that you supply to the template are not used as you think they are. For example, what if you added the following line to `main()` in Listing 21.6?

```
cout << Min( 'APPLE' , 'ORANGE' ) << endl;
```

If you don't think about what you're doing in the previous line, you might jump to the conclusion that the returned result will be `APPLE`. The truth is, however, that the preceding line might or might not give you the result that you expect. Why? Because the `"APPLE"` and `"ORANGE"` string constants result in pointers to `char`. This means that the program will compile just fine, with the compiler creating a version of `Min()` that compares `char` pointers. But there's a big difference between comparing two pointers and comparing the data to which the pointers point. If `"ORANGE"` happens to be stored at a lower address than `"APPLE"`, then the preceding call to `Min()` results in `"ORANGE"`.

A way to avoid this problem is to provide a specific replacement function for `Min()` that defines exactly how you want the two string constants to be compared. When you provide a specific function, the compiler uses that function instead of creating one from the template. Listing 21.7 is a short program that demonstrates this important technique. When the program needs to compare the two strings, it doesn't call a function created from the template but instead uses the specific replacement function.

---

**Note:**

The complete source code and executable file for the `Template2` application can be found in the `CHAP21\Template2` directory of this book's CD-ROM.

---



```
#include <iostream.h>
#include <string.h>

template<class Type>
Type Min(Type arg1, Type arg2)
{
    Type min;

    if (arg1 < arg2)
        min = arg1;
    else
        min = arg2;

    return min;
}

char* Min(char* arg1, char* arg2)
{
    char* min;

    int result = strcmp(arg1, arg2);

    if (result < 0)
        min = arg1;
    else
        min = arg2;

    return min;
}

int main()
{
    cout << Min(15, 25) << endl;
    cout << Min(254.78, 12.983) << endl;
    cout << Min('A', 'Z') << endl;
    cout << Min("APPLE", "ORANGE") << endl;

    return 0;
}
```

---

## Creating Class Templates

Just as you can create abstract functions with function templates, so, too, can you create abstract classes with class templates. A class template represents a class, which in turn represents an object. When you define a class template, the compiler takes the template and creates a class. You then instantiate objects of

the class. As you can see, class templates add another layer of abstraction to the concept of classes.

You define a class template much as you define a function template, by supplying the template line followed by the class's declaration, as shown in Listing 21.8. Notice that, just as with a function template, you use the abstract data types given as parameters in the template line in the body of the class to define member variables, return types, and other data objects.

#### **Listing 21.8** `lst21_08.cpp`—Defining a Class Template

---

```
template<class Type>
class CMyClass
{
protected:
    Type arg;

public:
    CMyClass(Type arg)
    {
        CMyClass::arg = arg;
    }

    ~CMyClass() {};
};
```

---

When you're ready to instantiate objects from the template class, you must supply the data type that will replace the template parameters. For example, to create an object of the CMyClass class, you might use a line like this:

```
CMyClass<int> myClass(15);
```

The previous line creates a CMyClass object that uses integers in place of the abstract data type. If you want the class to deal with floating-point values, you create an object of the class using something like this:

```
CMyClass<float> myClass(15.75);
```

For a more complete example, suppose that you want to create a class that stores two values and that has member functions that compare those values. Listing 21.9 is a program that does just that. First, the listing defines a class template called CCompare. This class stores two values that are supplied to the constructor. The class also includes the usual constructor and destructor, as well as member functions for determining the larger or smaller of the values or to determine whether the values are equal.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)
**Note:**

The complete source code and executable file for the Template3 application can be found in the CHAP21\Template3 directory of this book's CD-ROM.

**Listing 21.9 Template3.cpp—Using a Class Template**

```
#include <iostream.h>

template<class Type>
class CCompare
{
protected:
    Type arg1;
    Type arg2;

public:
    CCompare(Type arg1, Type arg2)
    {
        CCompare::arg1 = arg1;
        CCompare::arg2 = arg2;
    }

    ~CCompare() {}

    Type GetMin()
    {
        Type min;

        if (arg1 < arg2)
            min = arg1;
        else
            min = arg2;

        return min;
    }

    Type GetMax()
    {
        Type max;
```

```

        if (arg1 > arg2)
            max = arg1;
        else
            max = arg2;

        return max;
    }

    int Equal()
    {
        int equal;

        if (arg1 == arg2)
            equal = 1;
        else
            equal = 0;

        return equal;
    }
};

int main()
{
    CCompare<int> compare1(15, 25);
    CCompare<double> compare2(254.78, 12.983);
    CCompare<char> compare3('A', 'Z');

    cout << "THE COMPARE1 OBJECT" << endl;
    cout << "Lowest: " << compare1.GetMin() << endl;
    cout << "Highest: " << compare1.GetMax() << endl;
    cout << "Equal: " << compare1.Equal() << endl;
    cout << endl;

    cout << "THE COMPARE2 OBJECT" << endl;
    cout << "Lowest: " << compare2.GetMin() << endl;
    cout << "Highest: " << compare2.GetMax() << endl;
    cout << "Equal: " << compare2.Equal() << endl;
    cout << endl;

    cout << "THE COMPARE3 OBJECT" << endl;
    cout << "Lowest: " << compare3.GetMin() << endl;
    cout << "Highest: " << compare3.GetMax() << endl;
    cout << "Equal: " << compare3.Equal() << endl;
    cout << endl;

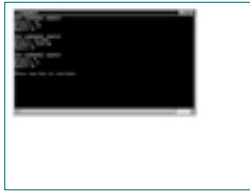
    return 0;
}

```

---

The main program instantiates three objects from the class template: one that

deals with integers, one that uses floating-point values, and one that stores and compares character values. After creating the three CCompare objects, main() calls the objects' member functions to display information about the data stored in each object. Figure 21.1 shows the program's output.



**FIG. 21.1** The Template3 program creates three different objects from a class template.

You can, of course, pass as many parameters as you like to a class template, just as you can with a function template. Listing 21.10 shows a class template that uses two different types of data.

#### **Listing 21.10** lst21\_10.cpp—Using Multiple Parameters with a Class Template

---

```
template<class Type1, class Type2>
class CMyClass
{
protected:
    Type1 arg1;
    Type2 arg2;

public:
    CMyClass(Type1 arg1, Type2 arg2)
    {
        CMyClass::arg1 = arg1;
        CMyClass::arg2 = arg2;
    }

    ~CMyClass() {}
};
```

---

To instantiate an object of the CMyClass class, you might use a line like this:

```
CMyClass<int, char> myClass(15, 'A');
```

Finally, you can use specific data types, as well as the placeholder data types, as parameters in a class template. You add the specific data type to the parameter list, just as you add any other parameter. Listing 21.11 is a short program that creates an object from a class template that uses two abstract parameters and one specific data type.



---

The complete source code and executable file for the Template4 application can be found in the CHAP21\Template4 directory of this book's CD-ROM.

---



## Listing 21.11 Template4.cpp—Using Specific Data Types as Parameters in a Class Template

---

```
#include <iostream.h>

template<class Type1, class Type2, int num>
class CMyClass
{
protected:
    Type1 arg1;
    Type2 arg2;
    int num;

public:
    CMyClass(Type1 arg1, Type2 arg2, int num)
    {
        CMyClass::arg1 = arg1;
        CMyClass::arg2 = arg2;
        CMyClass::num = num;
    }

    ~CMyClass() {}
};

int main()
{
    CMyClass<int, char, 0> myClass(15, 'A', 10);

    return 0;
}
```

---

## Using Run-Time Type Information

Run-Time Type Information (RTTI) was added to C++ so that programmers could obtain information about objects at runtime. This capability is especially useful when you're dealing with polymorphic objects, because it enables your program to determine at runtime what exact type of object it's currently working with. Later in this section, you'll see how important this type of information can be when you're working with class hierarchies. RTTI also can be used to safely downcast an object pointer. In this section, you'll discover how RTTI works and why you'd want to use it.

### Introducing RTTI

The RTTI standard introduces three new elements to the C++ language. The `dynamic_cast` operator performs downcasting (converting a base-class pointer to a derived-class pointer) of polymorphic objects; the `typeid` operator retrieves information (in the form of a `type_info` object) about an object; and the `type_info`

class stores information about an object, providing member functions that can be used to extract that information.

The public portion of the `type_info` class is defined in Visual C++ as shown in Listing 21.12.

#### **Listing 21.12** `lst21_12.cpp`—The `type_info` Class

---

```
class type_info {
public:
    virtual ~type_info();
    int operator==(const type_info& rhs) const;
    int operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const;
    const char* name() const;
    const char* raw_name() const;
};
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

As you can see, the class provides member functions that can compare objects for equality, as well as return the object's name, both as a readable text string and as a raw decorated object name.

## Performing Safe Downcasts

Once you start writing a lot of OOP programs, you'll run into times when you need to downcast one type of object to another. **Downcasting** is the act of converting a base-class pointer to a derived-class pointer (a **derived class** being a class that's derived from the base class). You use `dynamic_cast` to downcast an object, like this:

```
Type* ptr = dynamic_cast<Type*>(Pointer);
```

In the preceding example, `Type` is the type to which the object should be cast, and `Pointer` is a pointer to the object. If the pointer cannot be safely downcast, the `dynamic_cast` operator returns 0.

Suppose, for example, that you have a base class called `CBase` and a class derived from `CBase` called `CDerived`. Because you want to take advantage of polymorphism, you obtain a pointer to `CDerived`, like this:

```
CBase* derived = new CDerived;
```

Notice that, although you're creating a `CDerived` object, the pointer is of the base-class type, `CBase`. This is a typical scenario in programs that take advantage of OOP polymorphism.

Now suppose that you want to safely downcast the `CBase` pointer to a `CDerived` pointer. You might use `dynamic_cast`, as follows:

```
CDerived* ptr = dynamic_cast<CDerived*>(derived);
```

If `ptr` gets a value of 0, the downcast was not allowed because `derived` was not of the `CDerived` type (or of a type derived from `CDerived`).

## Getting Object Information

As I mentioned previously, you can use the `typeid` operator to obtain information about an object. Although the `dynamic_cast` operator applies only to polymorphic objects, you can use `typeid` on any type of data object. For example, to get information about the `int` data object, you can use lines like these:

```
const type_info& ti = typeid(int);
cout << ti.name();
```

In the first line, you can see that the `typeid` operator returns a reference to a `type_info` object. You can then use the object's member functions to extract information about the data object. In the preceding example, the `cout` object will output the word `int`. The `typeid` operator's single argument is the name of the data object for which you want a `type_info` object.

Of course, a better use for `typeid` is to compare and get information about classes that you have

defined in your program. Listing 21.13 is a short program that prints out information about the classes it defines.



The complete source code and executable file for the RTTI application can be found in the CHAP21\RTTI directory of this book's CD-ROM.

---

### **Listing 21.13 RTTI.cpp—Using the typeid Operator**

---

```
#include <iostream.h>
#include <typeinfo.h>

class CBase
{
public:
    CBase() {} ;
    ~CBase() {} ;
};

class CDerived : public CBase
{
public:
    CDerived() {} ;
    ~CDerived() {} ;
};

int main()
{
    CBase* base = new CBase;
    CBase* derived = new CDerived;

    const type_info& ti1 = typeid(CBase);
    const type_info& ti2 = typeid(CDerived);

    cout << "First object's name: " << ti1.name() << endl;
    cout << "First object's raw name: " << ti1.raw_name() << endl;
    cout << endl;

    cout << "Second object's name: " << ti2.name() << endl;
    cout << "Second object's raw name: " << ti2.raw_name() << endl;
    cout << endl;

    if (ti1 == ti2)
        cout << "The two objects are equal." << endl;
    else
        cout << "The two objects are not equal." << endl;

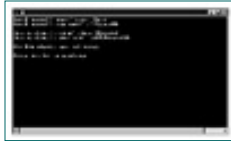
    cout << endl;

    delete base;
    delete derived;

    return 0;
}
```

---

Listing 21.13 first defines a base class called CBase. The program then derives a second class, called CDerived, from the base class. In main(), the program instantiates an object from each class, the objects being called base and derived. Then the program calls typeid to obtain type\_info objects for each class. Finally, the program calls the type\_info member functions to extract information about the classes, as well as to compare the classes for equality. Figure 21.2 shows the program's output. If you have trouble running the RTTI program shown in Listing 21.13, jump ahead to the next section, which tells you how to enable RTTI.



**FIG. 21.2** This is Listing 21.13 in action.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Preparing to Use RTTI

If you get a strange error message or warning when you try to compile the RTTI program, you probably don't have RTTI enabled. To enable RTTI, follow this procedure:

1. Choose the Project, Settings command from Developer Studio's menu bar. The Project Settings property sheet appears (see Figure 21.3).



**FIG. 21.3** The Project Settings property sheet contains many options.

2. Click the C/C++ tab. The C/C++ setting-options page appears, as shown in Figure 21.4.



**FIG. 21.4** The C/C++ options page contains language-specific project settings.

3. In the Category box, choose the C++ Language item. The C++ language options appear (see Figure 21.5).



**FIG. 21.5** The C++ language options include the RTTI settings.

4. Choose the Enable Run-Time Type Information (RTTI) option and then click OK to finalize your choices.

Also, be sure that you include the TYPEID.H header file in any source code file that calls the typeid operator. If you fail to do this, your program will not compile.

## A Common Use for RTTI

All of the previous discussion is okay from a nuts-and-bolts perspective, but why would you want to use RTTI in the first place? When you're writing programs that incorporate polymorphic objects, RTTI comes in handy. To see why, first take a look at Listing 21.14, which is a program that uses a simple case of polymorphism. The program defines two classes. The CBase class acts as the base class for the second class, CDerived. That is, CDerived is derived from CBase. In addition, CDerived overrides CBase's two virtual functions so that the functions perform as is appropriate for a Derived object.

---

**Note:**

The complete source code and executable file for the RTTI2 application can be found in the CHAP21\RTTI2 directory of this book's CD-ROM.

---

**Listing 21.14 RTTI2.cpp—A Typical Case of Class Derivation**

```
#include <iostream.h>
#include <typeinfo.h>

class CBase
{
public:
    CBase() {};
    ~CBase() {};

    virtual char* Func1() { return "CBase Func1()"; };
    virtual char* Func2() { return "CBase Func2()"; };
};

class CDerived : public CBase
{
public:
    CDerived() {};
    ~CDerived() {};

    char* Func1() { return "CDerived Func1()"; };
    char* Func2() { return "CDerived Func2()"; };
};

int main()
{
    CBase* base = new CBase;
    CBase* derived = new CDerived;

    cout << base->Func1() << endl;
    cout << base->Func2() << endl;
    cout << endl;

    cout << derived->Func1() << endl;
```

```

        cout << derived->Func2() << endl;
        cout << endl;

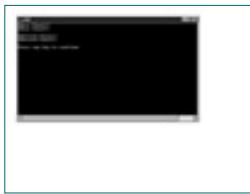
        delete base;
        delete derived;

        return 0;
}

```

---

In `main()`, the program creates an object from each class. But notice how pointers to both objects are of the `CBase` type. This is how polymorphism works. Although both pointers are of the `CBase` type, when the pointers are used to call the polymorphic `Func1()` and `Func2()` functions, the correct versions of the functions are called for the object type. You can tell from the program output, shown in Figure 21.6, that this is true. You’ve just witnessed polymorphism in action. Although both pointers are of the base-class type, the program automatically knows which set of member functions to call.



**FIG. 21.6** Listing 21.14’s output demonstrates polymorphism in action.

Now suppose that the `CBase` class’s `Func2()` member function is not virtual, but you still want to override the function in a derived class. As long as you’re using pointers to the specific class types—that is, the derived class’s pointer is of the derived class’s type rather than of the base class’s type—you won’t have any problem overriding the function. But what if you’re still using polymorphism in the program? Then, when you try to call the derived object’s `Func2()` member function, the base class’s `Func2()` is called instead. That keyword `virtual` sure makes a big difference, eh?

The solution to this dilemma is to use RTTI casting to downcast the class pointers whenever you need to be sure that the derived class’s version of the function is called rather than the base class’s version. Listing 21.15 is a program that demonstrates how this works. As in Listing 21.14, this program defines the `CBase` and `CDerived` classes. However, in this case, you’ve removed the `virtual` keyword from `CBase`’s `Func2()` member function. In other words, `Func2()` is no longer a polymorphic function.

---

**Note:**

The complete source code and executable file for the RTTI3 application can be found in the `CHAP21\RTTI3` directory of this book’s CD-ROM.

---

**Listing 21.15 RTTI3.cpp—Using Casting**

---



```

#include <iostream.h>

```



```

#include <typeinfo.h>

class CBase
{
public:
    CBase() {} ;
    ~CBase() {} ;

    virtual char* Func1() { return "CBase Func1()" ; };
    char* Func2() { return "CBase Func2()" ; };
};

class CDerived : public CBase
{
public:
    CDerived() {} ;
    ~CDerived() {} ;

    char* Func1() { return "CDerived Func1()" ; };
    char* Func2() { return "CDerived Func2()" ; };
};

int main()
{
    CBase* base = new CBase;
    CBase* derived = new CDerived;

    cout << base->Func1() << endl;
    cout << base->Func2() << endl;
    cout << endl;

    cout << "BEFORE CAST:" << endl;
    cout << derived->Func1() << endl;
    cout << derived->Func2() << endl;
    cout << endl;

    CDerived* ptr = dynamic_cast<CDerived*>(derived);

    cout << "AFTER CAST: " << endl;
    cout << ptr->Func1() << endl;
    cout << ptr->Func2() << endl;
    cout << endl;

    delete base;
    delete derived;

    return 0;
}

```

---

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Namespaces

Every programmer is already familiar with the concept of *namespaces*.

Basically, a namespace defines a scope in which duplicate identifiers cannot be used. For example, you already know that you can have a global variable named value and then also define a function with a local variable called value. Because the two variables are in different namespaces, your program knows that it should use the local value when inside the function and the global value everywhere else.

Namespaces, however, do not extend far enough to cover some very thorny problems. One example is duplicate names in external classes or libraries. This issue crops up when a programmer is using several external files within a single project. None of the external variables and functions can have the same name as other external variables or functions. To avoid this type of problem, third-party vendors frequently add prefixes or suffixes to variable and function names to reduce the likelihood of some other vendors using the same name.

Obviously, the C++ gurus have come up with a solution to such scope-resolution problems. Otherwise, we wouldn't be having this discussion! The solution is user-defined namespaces, about which you study in this section.

### Defining a Namespace

To accommodate user-defined namespaces, the keyword namespace was added to the C++ language. In its simplest form, a namespace is not unlike a structure or a class. You start the namespace definition with the namespace keyword, followed by the namespace's name and the declaration of the identifiers that will be valid within the scope of that namespace.

Listing 21.17 shows a namespace definition. The namespace is called A and includes two identifiers, i and j, and a function, Func(). Notice that the Func() function is completely defined within the namespace definition. You can also choose to define the function outside of the namespace definition, but in that case, you must prefix the function definition's name with the namespace's name, much as you would prefix a class's member-function definition with the class's name. Listing 21.18 shows this form of namespace function definition.

#### Listing 21.17 lst21\_17.cpp—Defining a Namespace

```
namespace A
```

```

{
    int i;
    int j;

    int Func()
    {
        return 1;
    }
}

```

---

### Listing 21.18 lst21\_18.cpp—Defining a Function Outside of the Namespace Definition

---

```

namespace A
{
    int i;
    int j;

    int Func();
}

int A::Func()
{
    return 1;
}

```

---

#### Tip:

Namespaces must be defined at the file level of scope or within another namespace definition. They cannot be defined, for example, inside of a function.

---

## Namespace Scope Resolution

Namespaces add a new layer of scope to your programs, but this means that you need some way of identifying that scope. The identification is, of course, the namespace's name, which you must use in your programs to resolve references to identifiers. For example, to refer to the variable `i` in namespace `A`, you write something like this:

```
A::i = 0;
```

You can, if you like, nest one namespace definition within another, as shown in Listing 21.19. In the case shown in the listing, however, you have to use more complicated scope resolutions to differentiate between the `i` variable declared in `A` and `B`, like this:

```
A::i = 0;
A::B::i = 0;
```

### Listing 21.19 lst21\_19.cpp—Nesting Namespace Definitions

---

```
namespace A
{
    int i;
    int j;

    int Func()
    {
        return 1;
    }

    namespace B
    {
        int i;
    }
}
```

---

If you're going to frequently reference variables and functions within namespace A, you can avoid using the A:: resolution by preceding the program statements with a using line, as shown in Listing 21.20.

### Listing 21.20 lst21\_20.cpp—Resolving Scope with the Using Keyword

---

```
using namespace A;
i = 0;
j = 0;
int num1 = Func();
```

---

## Unnamed Namespaces

Just to be sure that you're thoroughly confused, Visual C++ lets you have unnamed namespaces. You define an unnamed namespace exactly as you would any other namespace, except that you leave off the name. Listing 21.21 shows the definition of an unnamed namespace.

### Listing 21.21 lst21\_21.cpp—Defining an Unnamed Namespace

---

```
namespace
{
    int i;
    int j;

    int Func()
    {
        return 1;
    }
}
```

```
}
```

---

You refer to the identifiers in the unnamed namespace without any sort of extra scope resolution, like this:

```
i = 0;
j = 0;
int num1 = Func();
```

## Namespace Aliases

There might be times when you run into namespaces that have long names. In these cases, having to use that long name over and over in your program to access the identifiers defined in the namespace can be a major chore. To solve this problem, Visual C++ enables you to create *namespace aliases*, which are just replacement names for a namespace. You create an alias like this:

```
namespace A = LongName;
```

In the previous line of code, LongName is the original name of the namespace, and A is the alias. After the preceding line executes, you can access the LongName namespace, using either A or LongName. You can think of an alias as a nickname. If your name is Robert, you probably respond to Bob, too, right? Listing 21.22 is a short program that demonstrates namespace aliases.

---

### Note:

The complete source code and executable file for the Namespace application can be found in the CHAP21\Namespace directory of this book's CD-ROM.

---

## Listing 21.22 Namespace.cpp—Using a Namespace Alias

---



```
namespace ThisIsANamespaceName
{
    int i;
    int j;

    int Func()
    {
        return 2;
    }
}

int main()
{
    namespace ns = ThisIsANamespaceName;

    ns::i = 0;
    ns::j = 0;
```

```
int num1 = ns::Func();  
  
return 0;  
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

## Chapter 22

# Database Programming

- Learn about basic database concepts like records and fields
- Discover how the flat and relational database models differ
- Find out about MFC's ODBC database classes
- Learn to use AppWizard to create a basic ODBC application
- Learn to implement add, delete, update, sort, and filter abilities to your database program
- Discover the differences between using MFC's ODBC and DAO classes

Without a doubt, databases are one of the most popular computer applications. Virtually every business uses databases to keep track of everything from its customer list to the company payroll. Unfortunately, there are many different types of database applications, each of which defines its own file layouts and rules. Before the advent of Visual C++, programming database applications was a nightmare because it was up to the programmer to figure out all the intricacies of accessing the different types of database files.

Now, however, Visual C++ includes classes that are built upon the ODBC (Open Database Connectivity) and DAO (Data Access Objects) systems. Believe it or not, using AppWizard, you can create a simple database program without writing even a single line of C++ code. More complex tasks do require some programming but not as much as you might think.

This chapter, then, introduces you to programming with Visual C++'s ODBC classes. You also learn about the similarities and differences between ODBC and DAO. Along the way, you create a database application that can not only display records in a database, but also update, add, delete, sort, and filter records.

## Understanding Database Concepts

Before you can write database applications, you have to know a little about how databases work. Databases have come a long way since their invention, so there's much you can learn about them—much more than can be covered in an introductory chapter like this one. Limited space notwithstanding, in this section, you get a quick introduction to basic database concepts, as well as



learn about the two main types of databases: flat and relational.

## Using the Flat Database Model

Defined simply, a *database* is a collection of *records*. Each record in the database is comprised of *fields*. And each field contains information that's related to that specific record. For example, suppose you have an address database. In this database, you have one record for each person. Each record contains six fields: the person's name, street address, city, state, ZIP code, and phone number. So a single record in your database might look like this:

NAME: Ronald Wilson  
STREET: 16 Tolland Dr.  
CITY: Hartford  
STATE: CT  
ZIP: 06084  
PHONE: 860-555-3542

Your entire database will contain many records like the one just shown, each record containing information about a different person. To find a person's address or phone number, you search for his or her name. When you find that name, you also find all of the information that's included in the record with the name. This type of database system uses the flat database model. For home use or for small businesses, the simple flat database model can be a powerful tool. However, for large databases that must track dozens, or even hundreds, of fields of data, the relational database model is more appropriate.

## Using the Relational Database Model

A relational database is like several flat databases linked together. Using a relational database, you can not only search for individual records as you can with a flat database, but you can also relate one set of records to another. This enables you to store data much more efficiently. Each set of records in a relational database is called a table. The database management system (DBMS) can link these tables together in various ways by comparing keys that were defined by the person who created the database.

The example relational database that you'll use in this chapter was created using Microsoft Access. The database is a simple system for tracking employees, managers, and the departments for which they work. Figures 22.1, 22.2, and 22.3 show the tables that I defined when I created the database. The Employees table contains information about each of the store's employees; the Managers table contains information about each store department's manager; and the Departments table contains information about the departments themselves. (This database is very simple and probably not usable in the real world.)



**FIG. 22.1** The Employees table contains data fields for each store employee.



**FIG. 22.2** The Managers table contains data fields for each store department's manager.



**FIG. 22.3** The Departments table contains data about each store department.

## Accessing a Database

Relational databases are accessed using some sort of database scripting language. The most commonly used database language is SQL, which is used not only to manage databases on desktop computers, but also on the huge databases used by banks, schools, corporations, and other institutions with sophisticated database needs. Using a language like SQL, you can compare information in the various tables of a relational database and extract results that are made up of data fields from one or more tables combined.

Learning SQL, though, is a large task, one that is beyond the scope of this book (let alone this chapter). In fact, entire college-level courses are taught on the design, implementation, and manipulation of databases. In this chapter, you use the Employees table (refer to Figure 22.1) of the Department Store database in the sample database program that you'll soon develop. When you're done creating the application, you'll have learned one way to update the tables of a relational database without learning even a word of SQL.

## Discovering the MFC ODBC Classes

When you create a database program with Visual C++'s AppWizard, you end up with an application that draws extensively upon the various ODBC classes that have been incorporated into MFC. The most important of these classes are CDatabase, CRecordset, and CRecordView.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

AppWizard automatically generates the code needed to create an object of the CDatabase class. This object represents the connection between your application and the data source that you access. In most cases, using the CDatabase class in an AppWizard-generated program is transparent to you, the programmer. All of the details are handled by the framework.

AppWizard also generates the code needed to create a CRecordset object for the application. However, how you plan to use this object depends on whether you need to write code that calls the object's member functions. (Unless you're doing nothing more with the database than viewing its content, you need to call CRecordset member functions in your program.) The CRecordset object represents the actual data that's currently selected from the data source. The CRecordset object's member functions enable you to manipulate the data from the database in various ways.

Finally, the CRecordView object in your database program takes the place of the normal view window that you're accustomed to using in AppWizard-generated applications. A CRecordView window is special in that it's kind of like a dialog box that's being used as the application's display. This dialog box type of window retains a connection to the application's CRecordset object, hustling data back and forth between the program, the window's controls, and the record set. When you first create a new database application with AppWizard, it's up to you to add edit controls to the CRecordView window. These edit controls must be bound to the database fields that they represent, so the application framework knows where to display the data you want to view. In the next section, you see how these various database classes fit together, as you build the Employee application step by step.

## Creating an ODBC Database Program

Although creating a simple ODBC database program is easy with Visual C++, there are a number of steps that you must complete:

1. Register the database with the system.
2. Use AppWizard to create the basic database application.
3. Add code to the basic application to implement features not automatically supported by AppWizard.

In the following sections, you'll see how to perform the preceding steps as you create the Employee application, which enables you to add, delete, update, sort, and view records in the Employees table of the sample Department Store database.

## Registering the Database

Before you can create a database application, you must have the database that you want to access registered with the system. This process registers the selected tables in the database as data sources that you can access through the ODBC driver. Follow the steps that come next to accomplish this important task.

1. Create a folder called Database on your hard disk, and then copy the file named DeptStore.mdb from this book's CD-ROM to the new Database folder.



The DeptStore.mdb file is a database created with Microsoft Access. You use this database as the data source for the Employee application.

2. From the Windows 95 Start menu, run Control Panel. When Control Panel appears, double-click the 32-Bit ODBC icon. The ODBC Data Source Administrator property sheet appears (see Figure 22.4).



**FIG. 22.4** The ODBC Data Source Administrator property sheet holds several pages of options and information about data sources.

3. Select the User DSN tab, and then click the Add button. The Create New Data Source wizard appears. Select the Microsoft Access Driver from the list of drivers (see Figure 22.5), and click the Finish button.

The Microsoft Access Driver is now the ODBC driver that will be associated with the data source that you'll create for the Employee application.



**FIG. 22.5** The Create New Data Source wizard guides you through the steps needed to create a data source.

4. When the ODBC Microsoft Access 97 Setup dialog box appears, enter **Department Store** into the Data Source Name text box and then enter **Department Store Database** in the Description text box (see Figure 22.6).

The data source name is simply a way of identifying the specific data source that you're creating. The description field enables you to include more specific information about the data source.



**FIG. 22.6** The ODBC Microsoft Access 97 Setup dialog box enables you to name the database source.

5. Click the Select button. The Select Database file selector appears. Use the selector to locate and select the DeptStore.mdb file (see Figure 22.7).



**FIG. 22.7** The Select Database file selector enables you to locate the database file that you want to use.

6. Click OK to finalize the database selection, and then, in the ODBC Microsoft Access 7.0 Setup dialog box, click OK to finalize the data source creation process. Finally, click OK in the ODBC Data Source Administrator property sheet to dismiss the property sheet.

Your system is now set up to access the DeptStore.mdb database file with the Microsoft Access ODBC driver.

## Creating the Basic Employee Application

Now that you have your data source created and registered, it's time to create the basic Employee application. The steps that follow lead you through this process. After you've completed these steps, you'll have an application that can access and view the Employees table of the Department Store database.

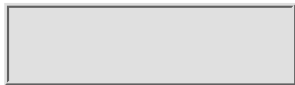
<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The complete source code and executable file for this part of the Employee application can be found in the CHAP22\Employee, Part 1 directory on this book's CD-ROM.



1. Choose File, New from Developer Studio's menu bar. The New property sheet appears (see Figure 22.8).



**FIG. 22.8** The New property sheet enables you to start many types of projects.

2. Make sure that MFC AppWizard (exe) is selected in the left pane. Then type **Employee** into the Project Name box, and click the Create button. The Step 1 dialog box appears (see Figure 22.9).



**FIG. 22.9** The Step 1 dialog box enables you to select an application type.

3. Choose Single Document, and then click the Next button. The Step 2 dialog box appears, as shown in Figure 22.10.

Choosing the Single Document option ensures that the Employee application will not allow more than one window to be open at a time.

4. Choose the Database View Without File Support option button, and then click the Data Source button. The Database Options dialog box appears (see Figure 22.11).



**FIG. 22.10** The Step 2 dialog box provides database support options.

By Choosing the Database View Without File Support option, you're telling AppWizard to automatically generate the classes you need to

view the contents of a database. This application will not use any supplemental files besides the database, so it doesn't need file (serializing) support.



**FIG. 22.11** The Database Options dialog box enables you to select a data source.

5. In the ODBC drop-down list, Choose the Department Store data source. Click the OK button, and the Select Database Tables dialog box appears, as shown in Figure 22.12.



**FIG. 22.12** The Select Database Tables dialog box enables you to select the tables that you want to associate with the application.

6. Choose the Employees table, and then click OK. The Step 2 dialog box reappears, filled in as shown in Figure 22.13.

You've now associated the Employees table of the Department Store data source with the Employee application.



**FIG. 22.13** After you select the data source, the Step 2 dialog box should look like this.

7. Click the Next button two times. The Step 4 dialog box appears (see Figure 22.14).



**FIG. 22.14** The Step 4 dialog box enables you to select application features.

8. Turn off the Printing and Print Preview option, and then click the Finish button to finalize your selections for the Employee application. The New Project Information dialog box that appears should look like Figure 22.15.





**FIG. 22.15** Your New Project Information dialog box should look like this.

9. Click the OK button, and AppWizard creates the basic Employee application.

You've now created the basic Employee application. At this point, you can compile the application by clicking the Build button on Developer Studio's toolbar, by choosing the Build, Build command from the menu bar, or by pressing F7 on your keyboard. After the program has compiled, choose the Build, Execute command from the menu bar—or press Ctrl+F5—to run the program. When you do, you see the window shown in Figure 22.16. You can use the database controls in the application's toolbar to navigate from one record in the Employees table to another. However, nothing appears in the window because you've yet to associate controls with the fields in the table that you want to view. You do that in the following section.



**FIG. 22.16** The basic Employee application looks nice but doesn't do much.

## Creating the Database Display

The next step in creating the Employee database application is to modify the form that displays data in the application's window. Because this form is just a special type of dialog box, it's easy to modify with Developer Studio's resource editor, as you'll discover while you complete the following steps.



The complete source code and executable file for this part of the Employee application can be found in the CHAP22\Employee, Part 2 directory on this book's CD-ROM.

1. Choose the ResourceView tab to display the application's resources (see Figure 22.17).



**FIG. 22.17** Click the ResourceView tab.

2. Open the resource tree by double-clicking the Employee Resources folder. Then open the Dialog resource folder the same way. Double-click the IDD\_EMPLOYEE\_FORM dialog box ID to open the dialog box into the resource editor (see Figure 22.18).

3. Click the static string in the center of the dialog box to select it, and then press Delete on your keyboard to remove the string from the dialog box.

4. Use the dialog box editor's tools to create the dialog box shown in Figure 22.19. When you create the edit boxes, use the following IDs: IDC\_EMPLOYEE\_ID, IDC\_EMPLOYEE\_NAME, IDC\_EMPLOYEE\_RATE, IDC\_EMPLOYEE\_DEPT. Also, set the Read-Only style (found on the Styles page of the Edit Properties property sheet) of the IDC\_EMPLOYEE\_ID edit box.

Each of these edit boxes will represent a field of data in the database. The first edit box is read-only because it will hold the database's primary key, which should never be modified.



**FIG. 22.18** Here, the dialog box is open in the resource editor.



**FIG. 22.19** Your final dialog box should look something like this.

5. Choose View, ClassWizard from the menu bar, or press Ctrl+W on your keyboard. The MFC ClassWizard property sheet appears, as shown in Figure 22.20. (Choose the Member Variables tab if necessary.)

6. With the IDC\_EMPLOYEE\_DEPT resource ID selected, click the Add Variable button. The Add Member Variable dialog box appears.

7. Click the arrow next to the Member Variable Name drop-down list, and then choose m\_pSet->m\_DeptID, as shown in Figure 22.21.



**FIG. 22.20** The MFC ClassWizard property sheet enables you to associate the form's controls with database fields.



**FIG. 22.21** The Add Member Variable dialog box is where you provide the actual association between a control and a database field.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

8. Associate other member variables (`m_pSet->m_EmployeeID`, `m_pSet->m_EmployeeName`, and `m_pSet->m_EmployeeRate`) with the edit controls in the same way. When you're done, the Member Variables page of the MFC ClassWizard property sheet should look like Figure 22.22.



**FIG. 22.22** You need to define member variables for each of the edit boxes.

By selecting member variables of the application's `CEmployeeSet` class (derived from MFC's `CRecordset` class) as member variables for the controls in the database view, you're establishing a connection through which data can flow between the controls and the data source.

9. Click the OK button in the MFC ClassWizard property sheet to finalize your changes.

You've now created a data display form for the Employee application. At this point, you can compile the application by clicking the Build button on Developer Studio's toolbar; by choosing the Build, Build command from the menu bar; or by pressing F7 on your keyboard. After the program has compiled, choose the Build, Execute command from the menu bar or press Ctrl+F5 to run the program. When you do, you see the window shown in Figure 22.23. As you can see, the application now displays the contents of records in the Employees database table. Use the database controls in the application's toolbar to navigate from one record in the Employees table to another.



**FIG. 22.23** The Employee application now displays data in its window.

After you've examined the database, try updating a record. To do this, simply change one of the record's fields (except the employee ID, which is the table's primary key and can't be edited). When you move to another record, the

application automatically updates the modified record. The commands in the application's Record menu also enable you to navigate through the records in the same manner as the toolbar buttons.

Notice that you've created a fairly sophisticated database access program without writing a single line of C++ code, a pretty amazing feat. Still, the Employee application is limited. For example, it can't add or delete records. As you might have guessed, that's the next piece of the database puzzle that you'll add.

## Adding and Deleting Records

As soon as you can add and delete records from a database table, you have a full-featured program for manipulating a flat (that is, not a relational) database. In this case, the "flat database" is the Employees table of the Department Store relational database. Adding and deleting records in a database table is an easier process than you might believe, thanks to Visual C++'s CRecordView and CRecordSet classes, which provide all of the member functions that you need to accomplish these common database tasks. Complete the following steps to include add and delete commands in the Employee application.



The complete source code and executable file for this part of the Employee application can be found in the CHAP22\Employee, Part 3 directory on this book's CD-ROM.

1. Choose the ResourceView tab, open the Menu folder, and double-click the IDR\_MAINFRAME menu ID. The menu editor appears, as shown in Figure 22.24.



**FIG. 22.24** Developer Studio's menu editor is in the right pane.

2. Open the Record menu in the editor, and then double-click the blank menu item at the bottom of the menu. The Menu Item Properties property sheet appears.
3. In the ID edit box, enter **ID\_RECORD\_ADD**, and in the Caption box, enter **&Add Record** (see Figure 22.25). When you press Enter, the menu editor adds the new command to the Record menu.



**FIG. 22.25** The Menu Item Properties property sheet enables you to create new menu items.

4. In the next blank menu item, add a delete command with the ID **ID\_RECORD\_DELETE** and the caption **&Delete Record**.
5. In the ResourceView pane, open the Toolbar folder, and then

double-click the IDR\_MAINFRAME ID. The application's toolbar appears in the resource editor.

6. Click the blank toolbar button to select it, and then use the editor's tools to draw the icon (it's supposed to be blue) shown in Figure 22.26 on the button.



**FIG. 22.26** This new button will control the Add function.

7. Double-click the new button in the toolbar. The Toolbar Button Properties property sheet appears. Choose ID\_RECORD\_ADD in the ID box (see Figure 22.27).



**FIG. 22.27** The Toolbar Button Properties property sheet enables you to associate the button with a menu item ID.

8. Create a new toolbar button that displays a red minus sign, giving the button the ID\_RECORD\_DELETE ID (see Figure 22.28). Move the Add and Delete buttons to a position before the Help (question mark) button.



**FIG. 22.28** The minus sign button will control the Delete function.

9. Choose the View, ClassWizard command from the menu bar, or just press Ctrl+W on your keyboard. The MFC Class Wizard property sheet appears. (Choose the Message Maps tab if necessary.)

10. Set the Class Name box to CEmployeeView, click the ID\_RECORD\_ADD ID in the Object IDs box, and then double-click COMMAND in the Messages box. The Add Member Function dialog box appears (see Figure 22.29).



**FIG. 22.29** The Add Member Function dialog box provides a default name for the new function.

11. Click the OK button to accept the default name for the new function. The function appears in the Member Functions box, as shown

in Figure 22.30.



**FIG. 22.30** The new functions appear in the Member Functions box.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

When OnMove() is called, the first thing that the program does is check the Boolean variable m\_bAdding to see whether the user is in the process of adding a new record:

```
if (m_bAdding)
```

If m\_bAdding is FALSE, the body of the if statement is skipped, and the else clause is executed. In the else clause, the program calls the base class's (CRecordView) version of OnMove(), which performs the default behavior for moving to the next record. If m\_bAdding is TRUE, the body of the if statement is executed. There, the program first resets the m\_bAdding flag:

```
m_bAdding = FALSE;
```

Then the program calls UpdateData() to transfer data out of the view window's controls:

```
UpdateData(TRUE);
```

A call to the record set's CanUpdate() method determines whether it's okay to update the data source, after which a call to the record set's Update() member function adds the new record to the data source:

```
if (m_pSet->CanUpdate())
    m_pSet->Update();
```

To rebuild the record set, the program must call the record set's Requery() member function, like this:

```
m_pSet->Requery();
```

A call to the view window's UpdateData() member function transfers new data to the window's controls:

```
UpdateData(FALSE);
```

Finally, the program sets the Employee ID field back to read-only so that the user can no longer change its contents:

```
CEdit* pCtrl = (CEdit*)GetDlgItem(IDC_EMPLOYEE_ID);
pCtrl->SetReadOnly(TRUE);
```



**Examining the *OnRecordDelete()* Function.** When the user clicks the Delete button (or chooses the Delete Record command from the Record menu), the *OnRecordDelete()* function gets called. In that function, deleting the record is just a matter of calling the record set's *Delete()* function:

```
m_pSet->Delete( );
```

As soon as the record is deleted, however, the program should display another record in its stead. That's where the call to the record set's *MoveNext()* function comes in:

```
m_pSet->MoveNext( );
```

The *MoveNext()* function moves the record set forward to the next record. A problem might arise, though, when the deleted record was in the last position or when the deleted record was the only record in the record set. A call to the record set's *IsEOF()* function will determine whether the record set was at the end. If the call to *IsEOF()* returns *TRUE*, the record set needs to be repositioned on the last record, which it is currently beyond. The record set's *MoveLast()* function takes care of this task. The code looks like this:

```
if (m_pSet->IsEOF( ))  
    m_pSet->MoveLast( );
```

When the last record has been deleted from the record set, the record pointer will be at the beginning of the set. The program can test for this situation by calling the record set's *IsBOF()* function. If this function returns *TRUE*, the program sets the current record's fields to *NULL*, like this:

```
m_pSet->SetFieldNull(NULL);
```

Finally, the last task is to update the view window's display:

```
UpdateData( FALSE );
```

## Sorting and Filtering

In many cases when you're accessing a database, you want to change the order in which the records are presented, or you might even want to search for records that fit certain criteria. MFC's ODBC database classes feature member functions that enable you to sort a set of records on any field. You can also call member functions to limit the records displayed to those whose fields contain given information, such as a specific name or ID. This latter operation is called **filtering**. In this section, you'll add sorting and filtering to the Employee application. Just follow the steps that come next.



The complete source code and executable file for this final part of the Employee application can be found in the CHAP22\Employee directory on this book's CD-ROM.

1. Add a Sort menu to the application's menu bar, as shown in Figure 22.32. Use the IDs *ID\_SORT\_ID*, *ID\_SORT\_NAME*, *ID\_SORT\_RATE*, and

ID\_SORT\_DEPARTMENT as the command IDs.



**FIG. 22.32** The Sort menu has four commands for sorting the database.

2. Use ClassWizard to add COMMAND functions for the four new sorting commands, using the function names suggested by ClassWizard. (Add the functions to the CEmployeeView class.) Figure 22.33 shows the resultant ClassWizard property sheet.



**FIG. 22.33** After you add the four new functions, ClassWizard should look like this.

3. Add a Filter menu to the application's menu bar. Use the IDs ID\_FILTER\_ID, ID\_FILTER\_NAME, ID\_FILTER\_RATE, and ID\_FILTER\_DEPT as the command IDs.
4. Use ClassWizard to add COMMAND functions for the four new filtering commands, using the function names suggested by ClassWizard. (Add the functions to the CEmployeeView class.)
5. Use Developer Studio's resource editor to create the dialog box shown in Figure 22.34. Give the edit control the ID, IDC\_FILTERVALUE.



**FIG. 22.34** Create this dialog box using the resource editor.

6. Start ClassWizard while the new dialog box is on the screen. The Adding a Class dialog box appears. Choose the Create a New Class option.
7. Click OK, and the New Class dialog box appears. In the Name box, type **CFilterDlg**, as shown in Figure 22.35, and then click OK.



**FIG. 22.35** The New Class dialog box enables you to associate a new C++ class with a dialog box element.

8. In the MFC ClassWizard property sheet, choose the Member Variables tab. Create a variable for the IDC\_FILTERVALUE control called

m\_filterValue, as shown in Figure 22.36. Click the OK button to dismiss ClassWizard.



**FIG. 22.36** Create a member variable for the edit control.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## ODBC versus DAO

In the previous section, you got an introduction to Visual C++'s ODBC classes and how they're used in an AppWizard-generated application. Visual C++ also features a complete set of DAO classes that you can use to create database applications. DAO is, in many ways, almost a superset of the ODBC classes, containing most of the functionality of the ODBC classes and adding a great deal of its own. Unfortunately, although DAO can read ODBC data sources for which ODBC drivers are available, it's not particularly efficient at the task. For this reason, the DAO classes are best suited for programming applications that manipulate Microsoft's .mdb database files, which are created by Microsoft Access. Other file formats that DAO can read directly are those created by FoxPro and Excel.

The DAO classes, which use the Microsoft Jet Database Engine, are so much like the ODBC classes, you can often convert an ODBC program to DAO simply by changing the class names in the program: CDatabase becomes CDaoDatabase, CRecordset becomes CDaoRecordset, and CRecordView becomes CDaoRecordView. One big difference between ODBC and DAO, however, is the way in which the system implements the libraries. ODBC is implemented as a set of DLLs, whereas DAO is implemented as OLE objects. Using OLE objects makes DAO a bit more up to date—at least as far as architecture goes—than ODBC.

Although DAO is implemented as OLE objects, you don't have to worry about dealing with those objects directly. The MFC DAO classes handle all of the details for you, providing data and function members that interact with the OLE objects. The CDaoWorkspace class provides more direct access to the DAO database engine object through static member functions. Although MFC handles the workspace for you, you can access its member functions and data members to explicitly initialize the database connection.

Another difference is that the DAO classes feature a more powerful set of methods that you can use to manipulate a database. These more powerful member functions enable you to perform sophisticated database manipulations without having to write a lot of complicated C++ code or SQL statements.

In summary, ODBC and DAO similarities are listed here:

- ODBC and DAO can both manipulate ODBC data sources. However, DAO is less efficient at this task, because it is best used with .mdb database files.
- AppWizard can create a basic database application based on either the

ODBC or DAO classes. Which type of application you want to create depends—at least in some part—on the type of databases with which you are working.

- ODBC and DAO both use objects of an MFC database class to provide a connection to the database being accessed. In ODBC, this database class is called `CDatabase`, whereas in DAO, the class is called `CDaoDatabase`. Although these classes have different names, the DAO database class contains some similar members to those found in the ODBC class.
- ODBC and DAO both use objects of a record set class to hold the currently selected records from the database. In ODBC, this record set class is called `CRecordset`, whereas in DAO, the class is called `CDaoRecordset`. Although these classes have different names, the DAO record set class contains, not only almost the same members as the ODBC class, but also a large set of additional member functions.
- ODBC and DAO use similar procedures for viewing the contents of a data source. That is, in both cases, the application must create a database object, create a record set object, and then call member functions of the appropriate classes to manipulate the database.

Some differences between ODBC and DAO are listed here:

- Although both ODBC and DAO MFC classes are similar (very similar, in some cases), some similar methods have different names. In addition, the DAO classes feature many member functions not included in the ODBC classes.
- ODBC uses macros and enumerations to define options that can be used when opening record sets. DAO, on the other hand, defines constants for this purpose.
- Under ODBC, snapshot record sets are the default, whereas under DAO, dynamic record sets are the default.
- The many available ODBC drivers make ODBC useful for many different database file formats, whereas DAO is best suited to applications that need to access only .mdb files.
- ODBC is implemented as a set of DLLs, whereas DAO is implemented as OLEobjects.
- Under ODBC, an object of the `CDatabase` class transacts directly with the data source. Under DAO, a `CDaoWorkspace` object sits between the `CDaoRecordset` and `CDaoDatabase` objects, thus enabling the workspace to transact with multiple database objects.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 23.1 Application Information and Management

Function	Description
AfxBeginThread()	Creates a new thread
AfxEndThread()	Terminates a thread
AfxGetApp()	Gets the application's CWinApp pointer
AfxGetAppName()	Gets the application's name
AfxGetInstanceHandle()	Gets the application's instance handle, which is the instance from which the default resources were loaded
AfxGetMainWnd()	Gets a pointer to the application's main window
AfxGetResourceHandle()	Gets the application's resource handle
AfxGetThread()	Gets a pointer to a CWinThread object
AfxRegisterClass()	Registers a window class in an MFC DLL
AfxRegisterWndClass()	Registers a Windows window class in an MFC application
AfxSetResourceHandle()	Sets the instance handle that determines where to load the application's default resources
AfxSocketInit()	Initializes Windows Sockets

ClassWizard Comment Delimiters

MFC defines a number of delimiters that ClassWizard uses to keep track of what it's doing, as well as to locate specific areas of source code. Although you'll rarely, if ever, use these macros yourself, you will see them embedded in your AppWizard applications, so you might like to know exactly what they do.

Table 23.2 fills you in.

[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

## Chapter 23

# Global Variables and Macros

- Discover the many macros defined by Visual C++
- Learn about MFC's global functions
- Find out about MFC's global variables

When you're writing programs, there are many types of data and operations that you must use again and again. Sometimes you have to do something as simple as creating a portable integer data type. Other times you need to do something a little more complex, like extracting a word from a long word value or storing the position of the mouse pointer. As you might know, Windows itself defines many constants and variables that you can use in your programs to help write programs faster. Using these previously defined constants and macros makes your programs more portable and more readable by other programmers. Besides the macros, global constants, and variables defined by Windows, MFC adds its own set. In the following tables, you'll get a look at the most important of these globally available constants, macros, and variables.

## Ten Categories of Macros and Globals

Because there are so many constants, macros, and globals, Visual C++ organizes its constants, macros, and globals into ten categories. Those categories are listed as follows. The following sections describe each of these categories and the symbols they define.

- Application information and management
- ClassWizard comment delimiters
- Collection class helpers
- CString formatting and message box display
- Data types
- Diagnostic services
- Exception processing
- Message maps
- Runtime object model services
- Standard command and window IDs

## Application Information and Management Functions



Because a typical Visual C++ application contains only one application object but many other objects created from other MFC classes, you frequently need to get information about the application in different places in a program. Visual C++ defines a set of global functions that return this information to any class in a program. These functions, which are listed in Table 23.1, can be called from anywhere within an MFC program. For example, you frequently need to get a pointer to an application's main window. The following function call accomplishes that task.

```
CWnd* pWnd = AfxGetMainWnd();
```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 23.2 ClassWizard Delimiters

Delimiter	Description
AFX_DATA	Starts and ends member variable declarations in header files that are associated with dialog data exchange
AFX_DATA_INIT	In a dialog class’s constructor, starts and ends dialog data exchange variable initialization
AFX_DATA_MAP	In a dialog class’s DoDataExchange() function, starts and ends dialog data exchange function calls
AFX_DISP	Starts and ends OLE Automation declarations in header files
AFX_DISP_MAP	Starts and ends OLE Automation mapping in implementation files
AFX_EVENT	Starts and ends OLE event declarations in header files
AFX_EVENT_MAP	Starts and ends OLE events in implementation files
AFX_FIELD	Starts and ends member variable declarations in header files that are associated with database record field exchange
AFX_FIELD_INIT	In a record set class’s constructor, starts and ends record field exchange member variable initialization
AFX_FIELD_MAP	In a record set class’s DoFieldExchange() function, starts and ends record field exchange function calls
AFX_MSG	Starts and ends ClassWizard entries in header files for classes that use message maps
AFX_MSG_MAP	Starts and ends message map entries
AFX_VIRTUAL	Starts and ends virtual function overrides in header files

Collection Class Helpers

Because certain types of data structures are so commonly used in programming, MFC defines collection classes that enable you to get these common data structures initialized quickly and manipulated easily. MFC includes collection classes for arrays, linked lists, and mapping tables. Each of these types of collections contain elements that represent the individual pieces of data that comprise the collection. In order to make it easier to access these elements, MFC defines a set of functions, shown in Table 23.3, that you can override for a particular data type.

Table 23.3 Collection Class Helper Functions

Function	Description
CompareElements()	Checks elements for equality
ConstructElements()	Constructs a new element (works similarly to a class constructor)
DestructElements()	Destroys elements (works similar to a class destructor)
DumpElements()	Provides diagnostic output in text form
HashKey()	Calculates hashing keys
SerializeElements()	Saves elements to or loads elements from an archive

## CString Formatting and Message Box Display

If you've done much Visual C++ programming, you know that MFC features a special string class, called `CString`, that makes string handling under C++ less cumbersome. `CString` objects are used extensively throughout MFC programs. Even when dealing with strings in a resource's string table, `CString` objects can come in handy, as the following global functions, which replace format characters in string tables, show (see Table 23.4). There's also a global function for displaying a message box.

**Table 23.4** `CString` Formatting and Message Box Functions

Function	Description
AfxFormatString1()	Replaces the format characters (i.e., %1) in a string resource with a given string
AfxFormatString2()	Replaces the format characters %1 and %2 in a string resource with the given strings
AfxMessageBox()	Displays a message box

## Data Types

The most commonly used constants are those that define a portable set of data types. You've seen tons of these constants, which are named with all uppercase letters, used in Windows programs. You'll recognize many of these from the Windows SDK. Others are included only as part of Visual C++. You use these constants exactly as you would any other data type. For example, to declare a Boolean variable, you'd write something like this:

```
BOOL flag;
```

Table 23.5 lists the most commonly used data types defined by Visual C++ for Windows 95 and NT.

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

[Brief](#)   [Full](#)

- ✚ [Advanced Search](#)
- ✚ [Search Tips](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 23.5 Commonly Used Data Types

Constant	Data Type
BOOL	Boolean value
BSTR	32-bit pointer to character data used with OLE
BYTE	8-bit unsigned integer
COLORREF	32-bit color value
DWORD	32-bit unsigned integer
LONG	32-bit signed integer
LPARAM	32-bit window-procedure parameter
LPCRECT	32-bit constant RECT structure pointer
LPCSTR	32-bit string-constant pointer
LPSTR	32-bit string pointer
LPVOID	32-bit void pointer
LRESULT	32-bit window-procedure return value
POSITION	The position of an element in a collection
UINT	32-bit unsigned integer
WNDPROC	32-bit window-procedure pointer
WORD	16-bit unsigned integer
LPARAM	32-bit window-procedure parameter

Diagnostic Services

After you have your program written, you’re far from done. Then comes the grueling task of testing, which means rolling up your sleeves, cranking up your debugger, and weeding out all the “gotchas” hiding in your code. Luckily, MFC provides many macros, functions, and global variables that you can use to incorporate diagnostic abilities into your projects. Using these tools, you can do everything from printing output to a debugging window to checking the integrity of memory blocks. Table 23.6 lists these valuable diagnostic macros, functions, and global variables.

Table 23.6 Diagnostic Macros, Functions, and Global Variables

Symbol	Description
AfxCheckMemory()	Verifies the integrity of allocated memory
AfxDoForAllClasses()	Calls a given iteration function for all classes that are derived from CObject and that incorporate run-time type checking
AfxDoForAllObjects()	Calls a given iteration function for all objects that were derived from CObject and that were allocated with the new operator

afxDump	A global CDumpContext object that enables a program to send information to the debugger window
AfxDump()	Dumps an object's state during a debugging session
AfxEnableMemoryTracking()	Toggles memory tracking
AfxIsMemoryBlock()	Checks that memory allocation was successful
AfxIsValidAddress()	Checks that a memory address range is valid for the program
AfxIsValidString()	Checks string pointer validity
afxMemDF	A global variable that controls memory-allocation diagnostics; can be set to allocMemDF, DelayFreeMemDF, or checkAlwaysMemDF
AfxSetAllocHook()	Sets a user-defined hook function that is called whenever memory allocation is performed
afxTraceEnabled	A global variable that enables or disables TRACE output
afxTraceFlags	A global variable that enables the MFC reporting features
ASSERT	Prints a message and exits the program if the assert expression is false
ASSERT_VALID	Validates an object by calling the object's AssertValid() function
DEBUG_NEW	Used in place of the new operator in order to trace memory-leak problems
TRACE	Creates formatted strings for debugging output
TRACE0	Same as TRACE but requires no arguments in the format string
TRACE1	Same as TRACE but requires one argument in the format string
TRACE2	Same as TRACE but requires two arguments in the format string
TRACE3	Same as TRACE but requires three arguments in the format string
VERIFY	Like ASSERT, but VERIFY evaluates the assert expression in both the Debug and Release versions of MFC—if the assertion fails, a message is printed, and the program is halted only in the Debug version

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

[Brief](#)   [Full](#)

- ✚ [Advanced Search](#)
- ✚ [Search Tips](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY IT KNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 23.7 Exception Macros and Functions**

Symbol	Description
AfxAbort()	Terminates an application upon a fatal error
AfxThrowArchiveException()	Throws an archive exception
AfxThrowDAOException()	Throws a CDaoException
AfxThrowDBException()	Throws a CDBException
AfxThrowFileException()	Throws a file exception
AfxThrowMemoryException()	Throws a memory exception
AfxThrowNotSupportedException()	Throws a not-supported exception
AfxThrowOleDispatchException()	Throws an OLE automation exception
AfxThrowOleException()	Throws an OLE exception
AfxThrowResourceException()	Throws a resource-not-found exception
AfxThrowUserException()	Throws an end user exception
AND_CATCH	Begins code that will catch specified exceptions not caught in the preceding TRY block
AND_CATCH_ALL	Begins code that will catch all exceptions not caught in the preceding TRY block
CATCH	Begins code for catching an exception
CATCH_ALL	Begins code for catching all exceptions
END_CATCH	Ends CATCH or AND_CATCH code blocks
END_CATCH_ALL	Ends CATCH_ALL code blocks
THROW	Throws a given exception
THROW_LAST	Throws the most recent exception to the next handler
TRY	Starts code that'll accommodate exception handling

**Note:**

The exception macros listed in Table 23.7 are included in MFC only to provide compatibility with earlier versions of the language. New programs should not use the macros, but rather the new exception-specific keywords such as try and catch.

- See “Understanding Exceptions,” p. 420

## Message Map Macros

Windows is an event-driven operating system, which means that every Windows application must handle a flood of messages that flow between an application and the system. MFC does away with the



clunky switch statements that early Windows programmers had to construct in order to handle messages and replaces those statements with a message map, which is nothing more than a table that matches a message with its message handler. In order to simplify the declaration and definition of these tables, MFC defines a set of message map macros. Many of these macros, which are listed in Table 23.8, will already be familiar to experienced MFC programmers.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



Brief    Full

[Advanced Search](#)

[Search Tips](#)



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Exception Processing

One of the newest elements of the C++ language is exceptions, which give a program greater control over how errors are handled. MFC increases the value of exceptions by defining a set of macros and functions that you can use to better handle errors in your applications. These macros and functions are listed in Table 23.7.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 23.8 Message Map Macros

Macro	Description
BEGIN_MESSAGE_MAP	Begins a message map definition
DECLARE_MESSAGE_MAP	Starts a message map declaration
END_MESSAGE_MAP	Ends a message map definition
ON_COMMAND	Begins a command-message message map entry
ON_COMMAND_RANGE	Begins a command-message message map entry that maps multiple messages to a single handler
ON_CONTROL	Begins a control notification message map entry
ON_CONTROL_RANGE	Begins a control notification message map entry that maps multiple control IDs to a single handler
ON_MESSAGE	Begins a user-message message map entry
ON_REGISTERED_MESSAGE	Begins a registered user-message message map entry
ON_UPDATE_COMMAND_UI	Begins a command-update message map entry
ON_UPDATE_COMMAND_UI_RANGE	Begins a command-update message map entry that maps multiple command-update messages to a single handler

Runtime Object Model Services

Frequently in your programs, you need access to information about classes at runtime. MFC supplies a macro for obtaining this type of information in a CRuntimeClass structure. In addition, the MFC application frameworks relies on a set of macros to declare and define runtime abilities (such as object serialization and dynamic object creation).If you’ve used AppWizard at all, you’ve seen these macros used in the generated source code files. If you’re an advanced MFC programmer, you might have even used these macros yourself. Table 23.9 lists the run-time macros and their descriptions.

Table 23.9 Runtime Services Macros

Macro	Description
DECLARE_DYNAMIC	Used in a class declaration to enable runtime class information access
DECLARE_DYNCREATE	Used in a class declaration to allow the class (derived from CObject) to be created dynamically; also, allows runtime class information access
DECLARE_OLECREATE	Used in a class declaration to allow object creation with OLE automation

DECLARE_SERIAL	Used in a class declaration to allow object serialization, as well as runtime class information access
IMPLEMENT_DYNAMIC	Used in a class implementation to enable runtime class information access
IMPLEMENT_DYNCREATE	Used in a class implementation to allow dynamic creation of the object and runtime information access
IMPLEMENT_OLECREATE	Used in a class implementation to enable object creation with OLE
IMPLEMENT_SERIAL	Used in a class implementation to allow object serialization and runtime class information access
RUNTIME_CLASS	Returns a CRuntimeClass structure for the given class

---



---

**Note:**

As you can see, the macros in Table 23.9 deal with run-time information. In addition to the macros, Run-Time Type Information (RTTI) was added to C++ so that programmers could obtain information about objects at runtime. This capability is especially useful when you're dealing with polymorphic objects, because it enables your program to determine at runtime what exact type of object it's currently working with.

---

- See "Using Run-Time Type Information," p. 434

## Standard Command and Window IDs

There are myriad standard messages that can be generated by a user of a Windows application. For example, whenever the user selects a menu command from a standard menu like File or Edit, the program sends a message. Each of these standard commands is represented by an ID. In order to relieve the programmer of having to define the dozens of IDs that are often used in a Windows application, MFC defines these symbols in a file called AFXRES.H. Some of these IDs have obvious purposes (for example, ID\_FILE\_OPEN), but many others are used internally by MFC for everything from mapping standard Windows messages to their handlers to defining string table IDs to assigning IDs to toolbar and status bar styles. There are far too many of these identifiers to list here. However, if you're interested in seeing them, just load the AFXRES.H file from your Visual C++ installation folder.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

[Brief](#)   [Full](#)

- ✚ [Advanced Search](#)
- ✚ [Search Tips](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 24.1 The MFC WinInet Classes**

Class	Description
CInternetSession	Creates and manages an Internet session. All MFC WinInet applications must create a CInternetSession object before using other WinInet classes.
CInternetConnection	Creates and manages an actual Internet connection. This is the base class for the more specific connection classes CFtpConnection, CGopherConnection, and CHttpConnection.(continues)
CFtpConnection	Creates and manages an FTP connection.
CGopherConnection	Creates and manages a gopher connection.
CHttpConnection	Creates and manages an HTTP connection.
CInternetFile	Enables remote access to files on an Internet server. This is the base class for the more specific WinInet file classes CGopherFile and CHttpFile.
CGopherFile	Manages interaction with a file on a gopher server.
CHttpFile	Manages interaction with a file on an HTTP server.
CFileFind	Enables a program to search for files. This is the base class for the more specific WinInet file search classes CFtpFileFind and CGopherFileFind.
CFtpFileFind	Enables a program to search for files on an FTP server.
CGopherFileFind	Enables a program to search for files on a gopher server.
CGopherLocator	Retrieves a Gopher locator from a gopher server.
CInternetException	Manages exceptions generated by the WinInet classes.

In the following sections, you get a closer look at many of the classes listed in Table 24.1 as you create Internet applications with the WinInet classes. First, you create an application that can download and display the HTML code for a Web page. Then, you create an application that can display the contents of an FTP server, as well as download files from that server.

# Programming a Web Application

Believe it or not, creating an application that can log onto a Web server and retrieve a Web page is one of the easiest things to do with the WinInet classes. I say that with one major caveat: Although such a simple Web application can download and display a Web page's HTTP document, it is not able to translate the HTTP document into an actual visual Web page. Doing this requires that your application be able to parse an HTTP document and create a display according to the commands that it finds in the HTTP document. That's clearly a major task, one that requires the functionality of a full-fledged Web browser.

Still, connecting to a Web server and downloading its home page as an HTTP document is a good way to get your feet wet with WinInet programming. That's because downloading a Web page is one of the most basic things you can do with the WinInet classes, so it provides a good introduction to MFC Internet programming. To create your first Internet application, complete the steps that follow.

The complete source code and executable file for the HTTPApp application can be found in the CHAP24\HTTPApp directory on this book's CD-ROM.

1. Create a new AppWizard project workspace called HTTPApp, as shown in Figure 24.1.



**FIG. 24.1** Create a new project workspace called HTTPApp.

2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 24.2.

Step 1: Single document

Step 2: Default settings

Step 3: Default settings

Step 4: Turn off all options except 3D Controls

Step 5: Default settings

Step 6: Default settings



**FIG. 24.2** Your New Project Information dialog box should look like this.

3. Use the resource editor to create the dialog box shown in Figure 24.3. Give the dialog box the ID IDD\_URLDLG, and give the edit control the ID IDC\_URL.



**FIG. 24.3** Use the dialog box editor to create this dialog box.


4. Press Ctrl+W on your keyboard. The Adding a Class dialog box appears (see Figure 24.4).




**FIG. 24.4** Create a class for the new dialog box.

5. Click OK to create a class for the new dialog box. The Create New Class dialog box appears. In the Name box, type **CURLDlg** (see Figure 24.5), and click the OK button.

[Previous](#) [Table of Contents](#) [Next](#)

 **SEARCH**  
ITKNOWLEDGE

[Brief](#) [Full](#)  
+ [Advanced Search](#)  
+ [Search Tips](#)

 **BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 24

# Programming Internet Applications with WinInet

- Learn how to connect to the Internet
- Understand how to download a Web page
- Discover how to connect to an FTP server
- Learn to download files from an FTP server

As the Internet becomes more and more integrated with our daily lives, the more important network programming becomes. Microsoft is well aware of the impact the Internet is having on many people's lives. They are also aware that Windows programmers are more anxious than ever to find a good way to create Internet applications. In response, Microsoft created WinInet, a library of functions that makes it easier to create Internet applications. The best news is that MFC 5.0 encapsulates the WinInet library into a number of classes that enable you to connect to the Internet with only a few lines of code. In this chapter, you explore the WinInet classes. Along the way, you learn to access Web pages and to transfer files using the FTP protocol.

## Introducing MFC's WinInet Classes

Using MFC's WinInet classes, you can create Internet applications that use high-level function calls, writing Internet connection and communication code in a similar fashion to MFC code you've written previously. The WinInet classes enable your application to take a general approach to Internet

programming or, if you like, to dig deeper into the classes to get more control over HTTP, FTP, and gopher sessions. The WinInet classes are listed in Table 24.1 along with their descriptions:

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

6. In the MFC ClassWizard property sheet, select the Member Variables tab, and create a string member variable called `m_url` for the `IDC_URL` edit box, as shown in Figure 24.6. Click OK to close the MFC ClassWizard property sheet.
7. Use the resource editor to delete the Edit menu from the application's menu bar.



**FIG. 24.5** Name the new class `CURLDlg`.



**FIG. 24.6** Create a member variable for the edit box.

8. Add a Connect menu to the application's menu bar. Give the menu one command called Make Connection with a command ID of `ID_CONNECT_MAKECONNECTION`, as shown in Figure 24.7.
9. Use ClassWizard to associate the `ID_CONNECT_MAKECONNECTION` command with the `OnConnectMakeconnection()` message response function, as shown in Figure 24.8. Make sure that you have `CHTTPAppView` selected in the Class Name box before you add the function.
10. Click the Edit Code button, and then add the lines shown in Listing 24.1 to the new `OnConnectMakeconnection()` function, right after the `TODO: Add your command handler code here comment`.



**FIG. 24.7** Create a Connect menu for the application.



**FIG. 24.8** Add the `OnConnectMakeconnection()` member function.

**Listing 24.1** `lst24_01.cpp`—Code for the New `OnConnectMakeconnection()` Function

---

```

CURLDlg dialog(this);
dialog.m_url = "<http://www.mcp.com>";

int result = dialog.DoModal();

if (result == IDOK)
{
    CString url = dialog.m_url;
    CInternetSession internetSession;

    try
    {
        CHttpFile* httpFile =
            (CHttpFile*)internetSession.OpenURL(url);
        httpFile->SetReadBufferSize(4096);

        for (int x=0; x<30; ++x)
            httpFile->ReadString(m_webPageLines[x]);

        httpFile->Close();
    }
    catch (CInternetException* pException)
    {
        m_webPageLines[0] = "CInternetException received!";
        pException->ReportError();
    }

    internetSession.Close();
    Invalidate();
}

```

---

**11.** While still in the HTTPAppView.cpp file, add the following lines near the top of the file, right after the #endif compiler directive:

```

#include <afxinet.h>
#include "urldlg.h"

```

These lines include in your program the header files that declare the WinInet and dialog classes, so that you can access those classes in the program.

**12.** Add the following lines to the view class's constructor, right after the comment TODO: add construction code here:

```

for (int x=0; x<30; ++x)
    m_webPageLines[x] = "";

```

These lines initialize the array of strings to empty strings. The content of the string array is displayed on the screen by the OnDraw() function. Eventually, the array of strings will hold the first thirty lines of an HTML document.

**13.** Add the following lines to the class's OnDraw() function, right after the TODO: add

draw code for native data here comment:

```
for (int x=0; x<30; ++x)
    pDC->TextOut(20, 20*x+20, m_webPageLines[x]);
```

These lines display the contents of the m\_webPageLines[] string array on the screen.

**14.** Load the HTTPAppView.h header file, and add the following lines to the class's declaration in the Attributes section, right after the line CHTTPAppDoc\* GetDocument():

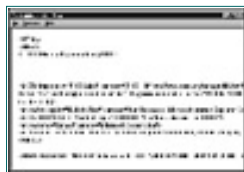
```
protected:
    CString m_webPageLines[30];
```

You've now completed the HTTPApp application. Select Developer Studio's Build, Build command to compile and link the application, and then select the Build, Execute command to run your new Internet application. When you do, the application's main window appears.

Select the Connect menu's Make Connection command to choose the Web URL to which you want to connect. This brings up the Make Connection dialog box, (see Figure 24.9). After entering the URL, select the OK button. HTTPApp will then attempt to connect to the URL and will display a maximum of 30 lines of the Web page's HTML document. Figure 24.10 shows HTTPApp connected to Microsoft's home page at [www.microsoft.com](http://www.microsoft.com).



**FIG. 24.9** Use the Make Connection dialog box to choose an URL.



**FIG. 24.10** This is HTTPApp connected to Microsoft's home page.

## Exploring the HTTPApp Application

You're probably surprised at how little programming it takes to connect to a Web site and download an HTML document. In this section, you dispel the mystery by examining the important parts of the HTTPApp application. But before looking at the code, take a look at the steps that you must complete to create a simple Web browser like HTTPApp. Those steps are listed as follows:

1. Create an object of the CInternetSession class.
2. Call the CInternetSession object's OpenURL() member function to connect to the selected URL, which creates an object of the CHttpFile class.
3. Call the CHttpFile object's ReadString() member function to read HTML lines from the remote file.
4. Process the HTTP lines to create the Web page display.
5. Call the CHttpFile object's Close() member function.
6. Call the CInternetSession object's Close() member function.

The following sections will examine how the preceding steps are implemented in the HTTPApp application. Most of the pertinent code is tucked away in the OnConnectMakeconnection() function, which MFC calls when the user selects the Connect, Make Connection command from the application's menu bar.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 24.2 Member Functions of the CInternetSession Class**

Function	Description
Close()	Closes an Internet session
EnableStatusCallback()	Enables a callback function, which is used for asynchronous operations
GetContext()	Gets the Internet session's context value
GetFtpConnection()	Establishes an FTP connection
GetGopherConnection()	Establishes a gopher connection
GetHttpConnection()	Establishes an HTTP connection
OnStatusCallback()	Updates an operation's status
OpenURL()	Connects to the given URL
QueryOption()	Provides error-checking services
ServiceTypeFromHandle()	Gets the service type from an Internet handle
SetOption()	Sets an Internet session's options

Notice that all of the calls to the WinInet classes' member functions are enclosed in a try program block. This is because, when connecting to an URL on the Internet, the risk of failure is high, especially when you consider that the user is typing the URL. For all you know, there's not even such an URL available. Another problem is timeouts, which might occur when the requested URL is currently unable to service the connection. Handling the WinInet exceptions, which are represented in MFC by the CInternetException class, is an important part of creating an Internet application with MFC.

## Reading the HTML Document

Once you have your HTTP connection established, you can start reading the HTML document. Due to some sort of bug in the MFC WinInet classes, however, you must first call the CHttpFile object's SetReadBufferSize() member function, which HTTPApp does like this:

```
httpFile->SetReadBufferSize(4096);
```

In the current version of the WinInet classes, if you fail to call SetReadBufferSize() before reading the file, you might get strange results.

---

### Note:

The SetReadBufferSize() problem appears to be fixed in Visual C++ 5.0. It doesn't hurt to set the buffer, however, so, to be on the safe side, it might be a good idea to leave the call to SetReadBufferSize() in the program.

---

Now you're ready to read the HTML file. You can do this in several ways, the easiest of which is calling the CHttpFile object's ReadString() member function. HTTPApp calls ReadString() from within a for loop, like this:

```
for (int x=0; x<30; ++x)
httpFile->ReadString(m_webPageLines[x]);
```

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full  
+ [Advanced](#)  
[Search](#)  
+ [Search Tips](#)

## Creating a *CInternetSession* Object

This first step couldn't be any easier, thanks to the fact that the *CInternetSession* class provides default values for all of the constructor's arguments—one of the advantages of programming WinInet through MFC. To create your *CInternetSession* object, just call the constructor like this:

```
CInternetSession internetSession;
```

Thanks to the default arguments supplied by the class, the preceding line is all you need to get started. If you want more control over the construction of your *CInternetSession* object, you can supply arguments for the constructor, whose signature follows:

```
CInternetSession(LPCTSTR pstrAgent = NULL, DWORD dwContext = 1,  
                DWORD dwAccessType = PRE_CONFIG_INTERNET_ACCESS,  
                LPCTSTR pstrProxyName = NULL, LPCTSTR pstrProxyBypass = NULL,  
                DWORD dwFlags = 0);
```

If you're interested in more advanced Internet applications, you can look up what these arguments mean in your Visual C++ online documentation. For most applications, though, the default arguments work just fine.

## Creating a *CHttpFile* Object

If you remember the list of steps presented earlier in the section "Exploring the HTTPApp Application," you know that the next step is to call the *CInternetSession* object's *OpenURL()* member function. Doing this not only connects the program to the given URL, but also supplies your application with an object of the *CHttpFile* class. You need the *CHttpFile* object to manipulate the remote file in several ways, not the least of which is to download each line of the HTTP file that the *CHttpFile* object represents. You call *OpenURL()* like this:

```
CHttpFile* httpFile = (CHttpFile*)internetSession.OpenURL(url);
```

Because *OpenURL()* actually returns an object of the *CInternetFile* type, the returned object must be cast to *CHttpFile*.

*OpenURL()* is only one of many member functions included in the *CInternetSession* class. All of the member functions are listed in Table 24.2 along with their descriptions.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 24.3 Member Functions of the CHttpFile Class

Function	Description
AddRequestHeaders()	Adds request headers to an HTTP request
Close()	Closes the CHttpFile object
GetFileURL()	Gets the file’s URL
GetObject()	Gets an HTTP request verb’s target object
GetVerb()	Gets a request verb
QueryInfo()	Gets response or request headers
QueryInfoStatusCode()	Gets an HTTP request’s status code
SendRequest()	Sends an HTTP request

Because it’s derived from CInternetFile, the CHttpFile class also gets a set of member functions from CInternetFile. Those member functions and their descriptions are listed in Table 24.4.

Table 24.4 Member Functions of the CInternetFile Class

Function	Description
Abort()	Closes the file and ignores all errors
Close()	Closes the file
Flush()	Flushes the file
Read()	Reads bytes from the file
ReadString()	Reads a stream of characters from the file
Seek()	Repositions the file pointer
SetReadBufferSize()	Sets the size of the read buffer
SetWriteBufferSize()	Sets the size of the write buffer
Write()	Writes bytes to the file
WriteString()	Writes a null-terminated string to the file

Processing the HTML Document

If you were writing a full-fledged Web browser, you’d need to take the text lines that the application is reading from the HTTP connection and translate them into an appropriate Web page display. Because HTML is a complex language, doing such a translation can be a major task. In fact, an entire encyclopedia could probably be dedicated to handling HTTP files. HTTPApp takes the easy way out, though, doing nothing more than displaying the HTML lines on the screen.

Closing the File and the Internet Session

When your application is finished with the file object, that object should be closed, like this:

```
httpFile->Close();
```

Closing the object frees its resources. The same is true of the CInternetSession object, which HTTPApp closes like this:

```
internetSession.Close();
```

## Programming an FTP Application

Although most people log onto the Internet through the World Wide Web, FTP servers provide a quick and easy way to download files. For that reason, this introduction to the WinInet classes includes a sample FTP application that enables you to not only browse the directory tree of an FTP server, but also to download files from the server.

Because an FTP application must keep track of directories and files, as well as provide an interface with which the user can browse the server, such an application is a bit more complex than the previous Web example. However, it's a heck of a lot easier to program an FTP application using WinInet than it is to build the application from scratch. Complete the steps below to build the FTPApp application.



The complete source code and executable file for the FTPApp application can be found in the CHAP24\FTPApp directory on this book's CD-ROM.

1. Create a new AppWizard project workspace called FTPApp as shown in Figure 24.11.



**FIG. 24.11** Create a new project workspace called FTPApp.

2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 24.12.

Step 1: Single document

Step 2: Default settings

Step 3: Default settings

Step 4: Turn off all options except 3D Controls

Step 5: Default settings

Step 6: Default settings



**FIG. 24.12** Your New Project Information dialog box should look like this.

3. Use the resource editor to create the dialog box shown in Figure 24.13. Give the dialog box the ID IDD\_FTPDLG and give the edit control the ID IDC\_FTP.

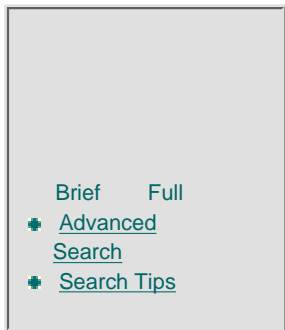




**FIG. 24.13** Use the dialog box editor to create this dialog box.

4. Click the ClassWizard button or press Ctrl+W on your keyboard. The Adding a Class dialog box appears (see Figure 24.14).

[Previous](#) [Table of Contents](#) [Next](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

The ReadString() function returns a CString object that holds the current text line from the HTML document. In HTTPApp, the program reads thirty lines from the HTML document and stores those lines in a CString array. The view class's OnDraw() function then displays the lines whenever the application's display must be repainted. (To see all of the lines at once, you might have to enlarge the application's window, which doesn't have a scroll bar.) You can, of course, call other member functions to manipulate a CHtmlFile object. Those member functions are listed in Table 24.3.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Table 24.5 Member Functions of the CFtpConnection Class

Function	Description
Close()	Closes the server connection
CreateDirectory()	Makes a directory on the server
GetCurrentDirectory()	Gets the connection’s current directory
GetCurrentDirectoryAsURL()	Gets the connection’s current directory as a URL
GetFile()	Gets a file
OpenFile()	Opens a file
PutFile()	Stores a file on the FTP server
Remove()	Removes a file
RemoveDirectory()	Removes a directory
Rename()	Renames a file
SetCurrentDirectory()	Sets the current directory on the server

Reading the FTP Server’s Root Directory

Getting the contents of the FTP server’s root directory requires several steps. FTPApp starts by calling the CFtpConnection object’s GetCurrentDirectory() member function:

```
ftpConnection->GetCurrentDirectory(m_curDirectory);
```

This line returns the current directory into the string given as the function’s single argument.

The program also creates a CFtpFileFind object to help it browse the files and directories on the FTP server:

```
CFtpFileFind ftpFileFind(ftpConnection);
```

The CFtpFileFind class’s constructor takes a pointer to a CFtpConnection object as its argument.

The program can now use the CFtpFileFind object to get the names and sizes of the files on the FTP server. To do this, the program starts the search by calling the CFtpFileFind object’s FindFile() member function:

```
ftpFileFind.FindFile();
```

When you start searching an FTP server’s directories, you must be sure to call FindFile() first; otherwise, subsequent calls to functions like FindNextFile() will fail.

Now that the search is started, the program can call the CFtpFileFind object’s FindNextFile() member function for each file or directory on the server:

```
gotFile = ftpFileFind.FindNextFile();
```

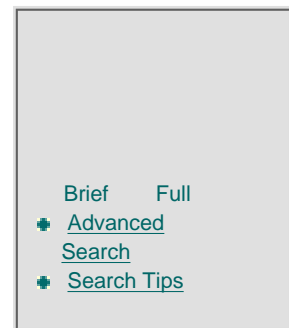
To get the complete contents of the current directory, the program must call FindNextFile() in a loop until the function returns zero, which indicates that the function call failed (in this case, due to there being no next file).

Each time the program calls FindNextFile(), it also calls the CFtpFileFind object's GetFileName() and GetLength() member functions, which return the current file's name and size:

```
m_displayStr[x] = ftpFileFind.GetFileName();  
m_fileLengths[x] = ftpFileFind.GetLength();
```

The CFtpFileFind class features only one other member function, GetFileURL(), which returns the URL of the current file. CFtpFileFind, however, inherits a set of member functions from its base class, CFileFind.

[Previous](#) [Table of Contents](#) [Next](#)



[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Exploring the FTPApp Application

Connecting to an FTP server is somewhat similar to connecting to a Web site. That is, there are common steps in both tasks, as you can see in the list that follows, which details what you must do to connect to an FTP server and get the contents of its root directory.

1. Create an object of the CInternetSession class.
2. Call the CInternetSession object's GetFtpConnection() member function to connect to the selected FTP server, which creates an object of the CFtpConnection class.
3. Call the CFtpConnection object's GetCurrentDirectory() member function to items in the server's root directory.
4. Create an CFtpFileFind object, which you'll use to browse the server's directories.
5. Call the CFtpFileFind object's FindFile() to start the file browsing process.
6. Repeatedly call the CFtpFileFind object's FindNextFile() and GetFileName() member functions to retrieve the directories and files stored in the current directory.
7. Delete the CFtpConnection object.
8. Call the CInternetSession object's Close() member function.

The following sections will examine how the preceding steps are implemented in the FTPApp application. In addition, you learn how to download files using the WinInet classes.

## Creating a CFtpConnection Object

After creating its CInternetSession object (which you already know how to do), FTPApp makes its FTP connection by calling the CInternetSession object's GetFtpConnection() member function, like this:

```
CFtpConnection* ftpConnection =  
    internetSession.GetFtpConnection(m_ftp);
```

The `GetFtpConnection()` method's single argument is a string containing the address of the FTP server (i.e. `ftp.mcp.com`) to which the program should connect. If the connection is made successfully, the function returns a pointer to a `CFtpConnection` object, which represents the FTP connection. If the function call fails, the return value is `NULL`. The previous code line is the easiest way to call `GetFtpConnection()`, which supplies default values for its five arguments. The function's full signature looks like this:

```
CFtpConnection* GetFtpConnection(LPCTSTR pstrServer,  
    LPCTSTR pstrUserName = NULL, LPCTSTR pstrPassword = NULL,  
    INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,  
    BOOL bPassive = FALSE); Throw (CInternetException);
```

If you're interested in gaining greater control over the creation of the `CFtpConnection` object, you can supply arguments for those with defaults. For more information, look up the function in your Visual C++ online documentation. Table 24.5 lists other member functions of the `CFtpConnection` class.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

## Browsing the FTP Server

Once the program has the contents of the server's root directory stored, the user can select a directory to browse. The user does this by double-clicking the directory, which causes MFC to call the application's `OnLButtonDblClk()` message response function. This function first converts the location that the user clicked into an index into the arrays that store the file name strings:

```
int index = (point.y - 60) / 20;
```

If index ends up as 0, the user has clicked on the previous directory command:

```
if (index == 0)
    PreviousDirectory();
```

Otherwise, if index is less than 20, it's a valid index into the file name array, `m_displayStr[]`. If the indexed string isn't empty, the user has clicked a directory, so the program can try to open that directory.

The first step in this process is to notify the user that the program has received the command. FTPApp does this by drawing a small rectangle next to the selected directory:

```
CClientDC clientDC(this);
clientDC.Rectangle(6, index*20+62, 14, index*20+74);
```

The program then starts an Internet session and builds a string containing the path to the requested directory:

```
CInternetSession internetSession;
CString dir = m_curDirectory + '/' + m_displayStr[index];
```

Now, the program can make the FTP connection:

```
CFTPConnection* ftpConnection =
    internetSession.GetFtpConnection(m_ftp);
```

With the connection in hand, the program sets the current server directory to the one selected by the user:

```
BOOLEAN dirOpen =
    ftpConnection->SetCurrentDirectory(dir);
```

If the directory opens successfully, the program stores the current directory in a member variable, creates a CFTPFileFind object, and starts a file search:

```
m_curDirectory = dir;  
CFTPFileFind ftpFileFind(ftpConnection);  
ftpFileFind.FindFile();
```

Finally, the program stores the new directory's contents, as shown in Listing 24.9.

---

**Listing 24.9 lst24\_09.cpp—Getting the Contents of the Current Directory**

---

```
BOOLEAN gotFile;  
int x = 1;  
  
do  
{  
    gotFile = ftpFileFind.FindNextFile();  
    m_displayStr[x] = ftpFileFind.GetFileName();  
    m_fileLengths[x] = ftpFileFind.GetLength();  
    ++x;  
}  
while ((x<20) && (gotFile));  
  
m_numFiles = x;
```

---

The program also, at this point, deletes the CFTPConnection object and closes the Internet connection:

```
internetSession.Close();  
delete ftpConnection;
```

## Downloading Files from an FTP Server

The last thing of interest in FTPApp is the way the program enables the user to download a file from the FTP server. The user indicates that he wants to download a file by right-clicking the file's name in the display. This action generates a WM\_RBUTTONDOWN Windows message that causes MFC to call the OnRButtonDown() message response function. In that function, the program uses the mouse-click location to calculate the index of the file name in the file name array, just as the program did in OnLButtonDblClk(). The program then starts an Internet session, sets the server's current directory, and finds the requested file. Again, you learned how to do all of this stuff when you studied the OnLButtonDblClk() member function.

If the server reports that the file exists, the application displays a message box, letting the user know that the program is about to download the file:

```
MessageBox("Click OK to download file");
```

The program then downloads the file by calling the CFTPConnection object's GetFile() member function:

```
ftpConnection->GetFile(file, file);
```

Here, the function's two arguments are the name of the file to download and the name of the downloaded copy of the file. Believe it or not, this one function call is all that's needed to transfer the file to the user's system. If you want more control over the downloading process, you can supply additional arguments for `GetFile()`. The function's full signature looks like this:

```
BOOL GetFile( LPCTSTR pstrRemoteFile, LPCTSTR pstrLocalFile,  
             BOOL bFailIfExists = TRUE,  
             DWORD dwAttributes = FILE_ATTRIBUTE_NORMAL,  
             DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,  
             DWORD dwContext = 1 );
```

As you can see, the additional parameters all have default values, which is why you don't need to supply them. The default value for the third argument, for example, tells `GetFile()` that, if the file named in the second argument already exists, the new version should overwrite the old one. If you supply this argument as `FALSE`, the call to `GetFile()` will fail if the file already exists. The fourth argument lets you set file attributes for the file; the third argument enables you to choose between a binary or ASCII transfer, and the last argument is a context identifier (which is used with Help files). For more details on these arguments, please refer to your Visual C++ online documentation.

Now that you know a little about WinInet, it's time to move on to Microsoft's newest Internet strategy, the ActiveX technologies. The next several chapters of this book are dedicated to this immense and important topic.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Part IV

# MFC and ActiveX

## Chapter 25

# An Introduction to ActiveX

- Learn why OLE was originally developed
- Discover how OLE grew to feature the sophisticated set of technologies it now supports
- Learn how COM provides the basis for OLE
- See how ActiveX grew from OLE 2
- Understand the different types of ActiveX programs

There's been a lot of confusion lately about what exactly comprises ActiveX. The documentation supplied by Microsoft is vague at best and often seems to contradict itself. A good definition (i.e., a definition that makes sense to us mere mortals) is almost impossible to come by. One almost has to wonder whether there's anyone at all who has a solid understanding of the ActiveX technologies. The truth is that ActiveX changes on almost a daily basis, making it hard at any given moment to nail down exactly what ActiveX is. In this chapter, though, you'll get an introduction to the areas of ActiveX that are most relevant to the topics in this book. By the end of this chapter, you should have a good grasp of what Microsoft is trying to accomplish with ActiveX.

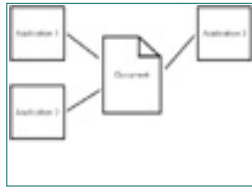
## OLE

A few years ago, Microsoft decided that it was time to come up with a better way to share data between applications. In answer to this need, Microsoft came up with something called OLE (Object Linking and Embedding). OLE implements two different ways of sharing and editing data between applications.

**Object linking** enables you to add a reference to another document from within your application's open document; in addition, many applications are able to link to the same document. For example, Figure 25.1 shows three applications linked to the same document. Because there is only one copy of the linked data, whenever that data changes, the contents of each of the documents that contain the linked data also change. You might, for example, have a chart from

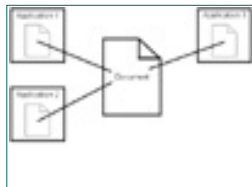


a spreadsheet program in two different business letters. If you change the chart, the contents of the chart in the two letters also change.



**FIG. 25.1** A linked document can be shared by several applications.

With **object embedding**, an actual copy of the source data is placed into the document (see Figure 25.2). This means that any application that embeds a particular set of data into its documents each has distinct copies. When you change the original, nothing happens to the copies in other documents. Likewise, if you change the embedded data, nothing happens to the original.



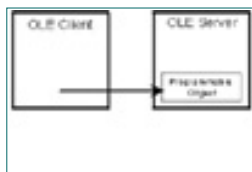
**FIG. 25.2** An embedded document is a copy of the source data.

## OLE 2

A while after creating OLE, Microsoft decided that OLE should be extended to enable applications to, not only share data, but also share functionality. Although Microsoft kept the name OLE for this new, and much more complex, technology, the letters are no longer an acronym, but rather just the word, OLE. That is, in OLE 2, “OLE” no longer stands for Object Linking and Embedding. This is because OLE 2 is so much more than just a way to share data.

OLE 2 enables you to share all kinds of objects between applications, including, not only data objects, but also **programmable objects**, which enable you to access the functionality of other applications. Programmable objects are created by an application’s programmers and then are exposed to the system when the application registers them in the system registry. After an object is registered, any other OLE-capable application can request access to the object and control it in almost the same manner as the application calls its own functions.

Controlling another application’s objects is called **OLE automation**. The application that supplies the programmable object is called an **OLE automation server**, whereas the application that accesses the object is called an **OLE automation client**. Figure 25.3 shows the relationship between an OLE automation server and client. Using OLE automation, you can write programs much faster, because you can call upon existing programmable objects to do tasks for which you’d otherwise have to write your own code. Similarly, you can write your own OLE automation servers, which can supply functionality not only to your own applications, but also to applications written by others.



**FIG. 25.3** OLE automation enables you to draw upon another application's functionality.

One of the most useful and common types of OLE objects is OLE controls. Although originally conceived as a way for programmers to create their own types of user-interface controls (such as custom buttons, gauges, and knobs), OLE controls quickly grew to include powerful mini-applications that could be embedded into applications. Microsoft's Visual Basic was the first programming environment to use anything like OLE controls. At that time, they were called Visual Basic controls (or VBX controls, for short). Soon, however, the technology improved, and OLE controls (called OCX controls, for short) were born.

## OLE and COM

COM (Component Object Model) is the system upon which OLE is built. COM is a specification for creating *binary objects* that can communicate with each other. You can think of a binary object as executable code that's loaded into the system and whose functions can be called by other executable code. DLLs, for example, are one type of binary object.

COM specifies a strict set of rules that programmers must follow when creating binary objects. These rules define a communications protocol that must be supported by all binary objects. By following these rules carefully, objects can be accessed and manipulated by any OLE-capable application and written in any language that supports OLE. Binary objects, too, can be written in any language, as long as the language supports the data structures needed to create the object according to the COM specification.

What are these COM rules and requirements? If you're really interested, you can pick up a copy of Kraig Brockschmidt's *Inside OLE*, published by Microsoft Press. This 1,200-page tome shows how to program OLE from the ground up. You could also pick up copies of Microsoft's OLE programmer's references, which, together (there are two volumes), offer several thousand more pages of OLE information.

Or, you could just be happy with knowing that OLE is built upon something called COM, and let MFC handle the rest.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

---

**Note:**

Programmers new to OLE often get COM and OLE mixed up. Simply put, COM is a specification that dictates how binary objects must be created and how they must communicate. The COM specification ensures that all binary objects in the system are compatible. OLE, on the other hand, describes a way in which COM objects can be used to share data and functionality.

---

## OLE and ActiveX

You can imagine the confusion OLE has caused as the technology has improved. First, OLE was meant only to provide a better way to share data between applications, using object linking and object embedding. Then, suddenly, a whole new dimension was added to OLE, as Microsoft improved the technology to include programmable objects. Now, yet again, Microsoft has extended OLE with a whole new set of abilities in an attempt to make the technology current with and applicable for Internet applications. To keep the name OLE at this point would have caused even more confusion, so Microsoft decided to go with a new name. ActiveX was born.

So, what exactly is ActiveX? Basically, ActiveX is an extended version of OLE 2 (OLE 3, if you will). Because ActiveX is built upon OLE 2, much of the terminology is transferable between the two. The following list shows how some common OLE terms translate into ActiveX terms:

- The OLE specification is now called the ActiveX specification
- OLE controls are now ActiveX controls
- OLE components are now ActiveX components
- OLE document objects are now ActiveX documents

More than just the terminology has changed, of course. For example, whereas OLE controls were designed to be used within a closed system, ActiveX controls can provide interactive and dynamic content to Web pages, which means that the concept of OLE controls has now been extended to include the entire Internet.

## ActiveX Today

As you've just discovered, ActiveX provides all the abilities of OLE, as well as extends those abilities with new features that better align the technology with Microsoft's vision for the Internet. Most importantly, ActiveX makes possible two new kinds of applications:

- Internet applications that can do real computing work, rather than just display text and images.
- General applications that have direct access to the Internet

A prime example of the first type of application in the previous list is a Web browser that can use ActiveX controls to embed mini-applications into Web pages. Such a Web page could contain ActiveX controls that do anything from play a game of Hangman to calculate a budget.

An example of the second type of application in the previous list might be a spreadsheet that can save its data anywhere on the Internet. When such applications become available, you'll be able to use the Internet as an actual extension of your local system, almost as if the Internet were a gigantic hard drive. Imagine almost instantly transferring a document directly from your word processor to another user's e-mail.

Microsoft isn't the only company struggling to harness the power of the Internet. Sun Microsystems is frantically developing its Java technology to fulfill many of the same goals. Java applets, for example, are similar to ActiveX controls in that they enable Web pages to contain mini-applications and so give Web browsers badly needed computing power. Java applets, however, are implemented very differently from ActiveX controls, and, although Java applets can be a powerful addition to Web pages, ActiveX is much wider in scope, giving many types of applications new Internet access abilities. Still, if you like, you can get the best of both worlds by using both ActiveX controls and Java applets in your Web pages.

## ActiveX and MFC

As you must have figured out by now, this book concentrates on getting the most out of Visual C++ 5.0's Microsoft Foundation Classes. Because MFC already encapsulates much of OLE 2, it also supports much of ActiveX. That is, not much has changed when it comes to the basic programming of ActiveX client and server applications and using automation to control programmable objects (now called ActiveX components).

The remaining chapters of this part of the book offer an introduction to the ActiveX basics. You'll learn to use AppWizard and MFC to create ActiveX clients (in this case, container applications), ActiveX servers, and ActiveX controls. You'll also learn about automation. The list below describes these important elements of ActiveX and OLE programming:

- ***ActiveX container applications*** enable the linking or embedding of data items into their own documents. When the user wants to edit the data item, the application might enable in-place editing, in which the application that created the data set merges its menus and toolbars with the container application's. Alternatively, the data item might enable editing by launching the application that created the data set.
- ***ActiveX server applications*** are the applications that create data items that can be embedded or linked into a container application's native document. When the user needs to edit an embedded or linked object, it is the server application that merges its menus and toolbars with the

container application's. In many cases, the user's request to edit an embedded or linked data item will launch an instance of the server application in its own window, rather than merge menus and toolbars.

- **Automation servers** contain programmable objects that can be accessed by a client application. After gaining access, the client application can use the programmable object's services by calling the object's functions. In this way, a client application can control (automate) a server application's objects.
- **ActiveX controls** are special types of binary objects that can be used as custom Windows controls. Although this description might bring to mind fancy buttons and gauges, ActiveX controls can be much more powerful than that and can be used to create mini-applications that supply additional computing power to their host applications.

In the following chapters, you'll discover how to use MFC to create all of the types of programs shown in the previous list. Along the way, you'll learn not only how to create your own ActiveX controls, but also how to embed them in your Web pages and so expose them to millions of Internet users.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 26

# Creating an ActiveX Container Application

- Learn how to create an ActiveX container application
- Discover how to embed and link objects
- Enable mouse selection of linked or embedded objects
- Learn how to select and delete ActiveX objects

An entire encyclopedia could be written on the programming of ActiveX applications. Obviously, it's not possible to fit an encyclopedia between the covers of this book. For that reason, Part IV of the book, including this chapter and the following three chapters, will give you an introduction to ActiveX programming. In this chapter, you look at the basic process required to create an ActiveX container application, which is an application that can hold ActiveX objects in its documents.

## Creating the Basic ActiveX Container Application

Thanks to AppWizard's amazing power, you can create a basic container application in only a couple of steps. The skeleton application created by AppWizard features all the basic functionality of an ActiveX container application, including the capability to embed, link, and edit ActiveX objects. Later in this chapter, in the sections entitled "Embedding an Object" and "Linking an Object," you'll see how to use the container application. But first you must create the application, which you can do by completing the following steps.

The complete source code and executable file for this version of the ActiveXCont1 application is located in the Chap26\ActiveXCont1, Part 1 folder on this book's CD-ROM.



1. Start a new AppWizard project workspace called ActiveXCont1, as shown in Figure 26.1.
2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 26.2.

Step 1: Single document

Step 2: Default settings

Step 3: Select Container and Yes, Please for OLE compound files.  
Leave other options as is.

Step 4: Default settings

Step 5: Default settings

Step 6: Default settings



**FIG. 26.1** Here's how you start ActiveXCont1.



**FIG. 26.2** These are the AppWizard settings for the ActiveXCont1 project.

You've now created the basic ActiveXCont1 application. To compile the program, select Developer Studio's Build, Build command. Then, select Build, Execute to run the program. When you do, you see the window shown in Figure 26.3. Thanks to AppWizard and MFC, the program is already a fairly powerful container application that can link or embed data supplied by an ActiveX server application.



**FIG. 26.3** The new ActiveXCont1 application's main window looks like this.

## Embedding an Object

To see ActiveX in action, select the application's Edit, Insert New Object command. When you do, the Insert Object dialog box appears (see Figure 26.4), from which you can select the type of object you want to insert and determine whether the object should be linked or embedded. The Create New option causes ActiveX to run the server application associated with the type of object you choose. You can then create a new object and embed it into ActiveXCont1's currently open document.



**FIG. 26.4** The Insert Object dialog box enables you to insert objects into a currently open document.



For example, leave the Create New option selected, and double-click the Bitmap Image in the Object Type box. ActiveX then requests the server to run so that you can create your new bitmap image. If you haven't changed the server associated with bitmap files in your system, Microsoft Paint will run and merge its menus and toolbars with those of the ActiveXCont1 container application. You can now use the new menus and tools to create the new bitmap image, all without ever having to leave the ActiveXCont1 application.

---

**Note:**

When a server application is capable of merging its menus and toolbars with the container application, it supports *in-place editing*. In-place editing means that you can edit the inserted object without having to leave the container application.

---

Normally, when you're finished creating the new bitmap image, you can click somewhere else in the document, and ActiveX removes the server application's menus and tools from the container application's window. This capability, however, has not yet been added to the ActiveXCont1 application. You'll take care of that missing piece of the puzzle later in the chapter, in the section entitled "Enabling Mouse Selection of Objects." For now, you can close the server application by pressing Esc. When you do, you'll see the ActiveXCont1 application with the new bitmap object embedded in its open document (see Figure 26.5).



**FIG. 26.5** The new bitmap image is embedded into the currently open document.

Because you created the bitmap object from scratch from inside the ActiveXCont1 application, the bitmap doesn't exist as a separate file on disk. Instead, it's an integral part of the document in which it's embedded. To see that this is true, select the application's File, Save command, and save the document under the file name Test.axc. Then, use Windows Explorer to examine the contents of the ActiveXCont1 project folder. You'll find the Test.axc file that you just saved, but you won't find a bitmap file. Because Test.axc contains the bitmap file, the bitmap image cannot be linked with other applications' documents. The bitmap is a separate entity that exists only in the Test.axc document.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Linking an Object

When you insert an object into a document, you can choose whether to create a new object, as you did in the previous section, or to insert an object that already exists on disk. A new object can only be embedded into the document, because the object doesn't exist as a disk file. Obviously, other applications can't link to a data object that doesn't exist separately on the disk.

Objects in their own files—such as bitmap files with the .BMP file extension—can be either linked or embedded into documents. To try this out, follow these steps:

1. Copy the Aztec.bmp file from the Chap26\ActiveXCont1 folder on this book's Web site to your ActiveXCont1 project folder. (Make sure you also turn off the file's read-only attribute.)
2. Select File, New from ActiveXCont1's menu bar. A new document appears in the window.
3. Select Edit, Insert New Object, and the Insert Object dialog box appears again.
4. Select the Create from File option. The dialog box changes to include a Browse button that you can select to find the file that you want to link or embed (see Figure 26.6).



**FIG. 26.6** When you want to insert a file, the Insert Object dialog box changes to give you file browsing capabilities.

5. Use the browse feature to locate and select the Aztec.bmp file you copied to the project's folder.

If you were to select the Insert Object dialog box's OK button at this point, the selected bitmap would be embedded into the document. However, you can link the bitmap by first selecting the Link option in the dialog box. When you do, click the dialog box's OK button, and ActiveX inserts the bitmap into the document by linking to the bitmap's file, as shown in Figure 26.7. Save the document containing the newly linked object under the name Test2.axc, and then close the document window by selecting File, New.

You might remember that when a linked object is edited, all documents that

contain a link to that object are automatically updated. To prove this, bring up Windows Explorer, and double-click the Aztec.bmp to call up the program you use to edit bitmaps. Now, change the bitmap any way you like, and save the bitmap back to disk.

To see how basic linking works, reload the Test2.ax1 file. Then, select the Edit, Links command. The Links dialog box, which displays the bitmap's link to the document, appears (see Figure 26.8). Click the Uppdate Now button to update the linked image. The document now shows the changes you made to the bitmap.



**FIG. 26.7** Here's the bitmap linked (rather than embedded) into the ActiveX1 document.



**FIG. 26.8** The Links dialog box enables the user to manipulate an item's link to the document.

---

**Note:**

If you edit an item from within the container application, the updating mechanism works automatically. For example, you could have edited the Aztec.bmp file by selecting the Edit, Linked Bitmap Image Object, Edit command. In that case, when you closed the bitmap editor, the changes would appear automatically in the container application's document.

---

## Understanding the ActiveXCont1 Skeleton Application

As you might have guessed, to implement the ActiveX functionality supported by the application, there's a lot going on under the hood of ActiveXCont1. In this section, you'll examine the pertinent parts of the AppWizard-generated code that's important to ActiveX.

---

**Note:**

As you look through MFC's ActiveX code, you'll notice that the word OLE comes up a lot. This is because MFC's classes and member functions were named before OLE was changed to ActiveX.

---

### Exploring the CActiveXCont1App Class

To understand the source code, the first place to look is in InitInstance(), where

the application performs its initialization at start-up. That function, which is defined as part of the CActiveXCont1App class, performs its normal initialization, as well as some special initialization for ActiveX. To initialize the ActiveX libraries, InitInstance() calls AfxOleInit(), as shown in Listing 26.1:

**Listing 26.1** lst26\_01.cpp—Calling AfxOleInit()

---

```
// Initialize OLE libraries
if (!AfxOleInit())
{
    AfxMessageBox( IDP_OLE_INIT_FAILED );
    return FALSE;
}
```

---

The InitInstance() function also calls the AfxEnableControlContainer() global function, which enables the application to act as a container for ActiveX controls. That function call looks like this:

```
AfxEnableControlContainer( );
```

Mixed in amongst the standard MFC function calls for setting up the application's document and view, InitInstance() calls the CDocTemplate object's SetContainerInfo() member function to give the application the menu and other resources it needs to handle an embedded object when said object is being edited in-place. That function call looks like this:

```
pDocTemplate->SetContainerInfo( IDR_CNTR_INPLACE );
```

If you look at the ActiveXCont1 application's menu resources, you'll see that there are two menus defined, with the IDs IDR\_CNTR\_INPLACE and IDR\_MAINFRAME. Those menus are combined to provide the container with normal and ActiveX commands on the menu bar.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

**SEARCH**  
 ITKNOWLEDGE

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Exploring the CActiveXCont1CntrlItem Class

Like everything else in MFC, an embedded or linked object is represented by an instance of a class. MFC provides the COleClientItem class to represent these types of objects. In the ActiveXCont1 application's source code, the CActiveXCont1CntrlItem class is derived from COleClientItem and so represents embedded objects for this specific application. Table 26.1 lists the functions provided by AppWizard for the CActiveXCont1CntrlItem class, along with their descriptions:

**Table 26.1 Member functions of the CActiveXCont1CntrlItem Class**

Function	Description
OnActivate()	Called by MFC when the object is activated in-place, but before the server's user interface (menus and tools) have been merged with the container application's interface. The AppWizard-generated version of this function gets a pointer to the activated item, checking to be sure that the pointer isn't NULL and that the pointer isn't equal to the this pointer. The function then calls the base class version of OnActivate().
OnChange()	Called by MFC whenever the user changes the embedded object, including editing, saving, or closing the object. The AppWizard version of this function simply calls the base class's OnChange() and then updates all views.
OnChangeItemPosition()	Called by MFC when the server requests a change to the in-place window's position. The AppWizard-generated version of this function simply calls the base class's version.
OnDeactivateUI()	Called by MFC when an object is deactivated so that the container application can restore its menus and tools. The AppWizard-generated version of this function performs most of its work by calling the base class version.
OnGetItemPosition()	Called by MFC to get the location of the object. The AppWizard-generated version of the function specifies a hard-coded position rectangle of 10, 10, 210, 210.
Serialize()	Called by MFC to store the object. The AppWizard-generated version does little more than call the base class version of the function.

## Exploring the CActiveXCont1View Class

When the user selects the Insert New Object command, MFC calls the view class's OnInsertObject() member function. OnInsertObject(), shown in Listing 26.2, first displays the Insert Object dialog box to gather information about the new object from the user. The function then displays the hourglass cursor and creates an instance of the CActiveXCont1CntrlItem class, which represents the object about to be

inserted into the document. If this new object is not a file, OnInsertObject() starts the appropriate server application to enable the user to edit the new object. Because a new object is always in a selected state, the function sets the m\_pSelection data member, which holds a pointer to the currently selected object, to the new object's pointer. Finally, the function updates all views and turns off the hourglass cursor.

---

**Note:**

The program line `pItem = new CActiveXContlCntrItem(pDoc)` only creates the container item—it doesn't initialize it. The info needed to initialize the container item from the Insert Object dialog box is contained in the `COleInsertDialog` dialog box. That's why the function that initializes the container item class is `dlg.CreateItem()`, a member of `COleInsertDialog`, not a member of `CActiveXContlView` or `CActiveXContlCntrItem`.

---

---

**Listing 26.2 lst26\_02.cpp—Responding to the Insert New Object Command**

---

```
void CActiveXContlView::OnInsertObject()
{
    // Invoke the standard Insert Object dialog box
    // to obtain information
    // for new CActiveXContlCntrItem object.
    COleInsertDialog dlg;
    if (dlg.DoModal() != IDOK)
        return;

    BeginWaitCursor();

    CActiveXContlCntrItem* pItem = NULL;
    TRY
    {
        // Create new item connected to this document.
        CActiveXContlDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pItem = new CActiveXContlCntrItem(pDoc);
        ASSERT_VALID(pItem);

        // Initialize the item from the dialog data.
        if (!dlg.CreateItem(pItem))
            AfxThrowMemoryException(); // any exception will do
        ASSERT_VALID(pItem);

        // If item created from class list (not from file) then launch
        // the server to edit the item.
        if (dlg.GetSelectionType() == COleInsertDialog::createNewItem)
            pItem->DoVerb(OLEIVERB_SHOW, this);

        ASSERT_VALID(pItem);

        // As an arbitrary user interface design,
        // this sets the selection
        // to the last item inserted.

        // TODO: reimplement selection as appropriate
        // for your application

        m_pSelection = pItem; // set selection to last inserted item
        pDoc->UpdateAllViews(NULL);
    }
}
```

```

        CATCH(CException, e)
        {
            if (pItem != NULL)
            {
                ASSERT_VALID(pItem);
                pItem->Delete();
            }
            AfxMessageBox(IDP_FAILED_TO_CREATE);
        }
    END_CATCH

    EndWaitCursor();
}

```

---

The `IsSelected()` member function (Listing 26.3) has an easy task—determine whether the given object is currently selected. The function only has to compare the given pointer to the `m_pSelection` data member, which holds a pointer to the currently selected item. If no item is selected, `m_pSelection` is `NULL`.

---

**Listing 26.3** `lst26_03.cpp`—The `IsSelected()` Member Function

---

```

BOOL CActiveXContlView::IsSelected(const CObject* pDocItem) const
{
    // The implementation below is adequate if your selection consists of
    // only CActiveXContlCntrItem objects. To handle different selection
    // mechanisms, the implementation here should be replaced.

    // TODO: implement this function that tests for a
    //       selected OLE client item

    return pDocItem == m_pSelection;
}

```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

Whenever the user resizes a document window when an object is being edited in-place, the server application needs to be informed of the change. This happens in the view class's `OnSize()` member function (Listing 26.4). The AppWizard-generated version of this function first calls the base class's `OnSize()`. It then gets a pointer to the currently active item. If the pointer is `NULL`, there is no active object. If the pointer is not `NULL`, the function calls `SetItemRects()` on behalf of the object, which sets the object's new bounding rectangle.

#### Listing 26.4 `lst26_04.cpp`—The `OnSize()` Member Function

```
void CActiveX1View::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->SetItemRects();
}
```

When a window containing an object that's being edited in-place receives the focus, the object being edited needs to take over the focus. This task is handled in the `OnSetFocus()` member function (Listing 26.5). This function first gets a pointer to the object being activated in-place. If the returned pointer is not `NULL` and the object's server has activated its user interface (merged its menus and tools), `OnSetFocus()` gets a pointer to the in-place window and gives the window the focus. Before returning, `OnSetFocus()` calls the base class's version of `OnSetFocus()`.

#### Listing 26.5 `lst26_05.cpp`—The `OnSetFocus()` Member Function

```
void CActiveX1View::OnSetFocus(CWnd* pOldWnd)
{
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL &&
        pActiveItem->GetItemState() == COleClientItem::activeUIState)
    {
        // need to set focus to this item if it is in the same view
        CWnd* pWnd = pActiveItem->GetInPlaceWindow();
        if (pWnd != NULL)
        {
            pWnd->SetFocus();    // don't call the base class
            return;
        }
    }

    CView::OnSetFocus(pOldWnd);
}
```

The last member function of interest in the view class is `OnDraw()` (Listing 26.6), which is responsible for drawing, not only the document's native data (data that's created by the application itself, not by ActiveX objects), but also any embedded or linked objects present in the document. As the comments in the function indicate, the AppWizard-generated code in `OnDraw()` is only temporary and must be replaced with your own code. The default function draws a single linked or embedded object by calling the object's `Draw()` member function.

---

**Listing 26.6** `lst26_06.cpp`—The `OnDraw()` Member Function

---

```
void CActiveXCont1View::OnDraw(CDC* pDC)
{
    CActiveXCont1Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
    // TODO: also draw all OLE items in the document

    // Draw the selection at an arbitrary position.
    // This code should be
    // removed once your real drawing code is implemented.
    // This position
    // corresponds exactly to the rectangle returned by
    // CActiveXCont1CntrItem,
    // to give the effect of in-place editing.

    // TODO: remove this code when final draw code is complete.

    if (m_pSelection == NULL)
    {
        POSITION pos = pDoc->GetStartPosition();
        m_pSelection =
            (CActiveXCont1CntrItem*)pDoc->GetNextClientItem(pos);
    }
    if (m_pSelection != NULL)
        m_pSelection->Draw(pDC, CRect(10, 10, 210, 210));
}
```

---

### Exploring the *CActiveXCont1Doc* Class

The application's document class also contains some ActiveX code. First, the document class is derived from `COleDocument`, like this:

```
class CActiveXCont1Doc : public COleDocument
```

As you might have guessed, documents of the `COleDocument` class can contain ActiveX objects.

The document class also implements the message map for ActiveX items on the application's Edit menu. Listing 26.7 shows how this message map appears in the AppWizard-generated code.

---

**Listing 26.7** `lst26_07.cpp`—The Document Class's Message Map

---

```
BEGIN_MESSAGE_MAP(CActiveXCont1Doc, COleDocument)
    //{AFX_MSG_MAP(CActiveXCont1Doc)
    // NOTE - the ClassWizard will add and remove
```

```

        // mapping macros here.
        // DO NOT EDIT what you see in these blocks
        // of generated code!
    //}}AFX_MSG_MAP
    // Enable default OLE container implementation
    ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE,
        ColeDocument::OnUpdatePasteMenu)
    ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE_LINK,
        ColeDocument::OnUpdatePasteLinkMenu)
    ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_CONVERT,
        ColeDocument::OnUpdateObjectVerbMenu)
    ON_COMMAND(ID_OLE_EDIT_CONVERT,
        ColeDocument::OnEditConvert)
    ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_LINKS,
        ColeDocument::OnUpdateEditLinksMenu)
    ON_COMMAND(ID_OLE_EDIT_LINKS, ColeDocument::OnEditLinks)
    ON_UPDATE_COMMAND_UI(ID_OLE_VERB_FIRST,
        ColeDocument::OnUpdateObjectVerbMenu)
END_MESSAGE_MAP()

```

---

Finally, the document class's constructor is responsible for enabling compound documents, as shown in Listing 26.8. Simply put, compound documents are documents that contain, not only data native to the application, but also linked or embedded ActiveX objects. Obviously, such a document requires a special type of storage. This special type of storage is called *structured storage*.

#### Listing 26.8 lst26\_08.cpp—The Document Class's Constructor

---

```

CActiveXCont1Doc::CActiveXCont1Doc()
{
    // Use OLE compound files
    EnableCompoundFile();

    // TODO: add one-time construction code here
}

```

---

Now that you have a general idea of what goes on inside the basic AppWizard-generated container application, it's time to extend the application to include some other important ActiveX features. In the next section, you'll learn how to enable the user to select and activate an ActiveX object with a mouse click.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.  
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Enabling Mouse Selection of Objects

In a standard container application, the user can select a linked or embedded object by clicking it with the mouse. Moreover, a double-click should activate the object for editing. Currently, the ActiveXCont1 application is unable to perform either of these functions. Perform the following steps to add the code needed to enable mouse selection of ActiveX objects.

The complete source code and executable file for this version of the ActiveXCont1 application is located in the Chap26\ActiveXCont1, Part 2 folder on this book's CD\_ROM.



1. Load the CntrItem.h file, and add the following line to the class's public Attributes section, right after the public keyword:

```
CRect m_itemRect;
```

This CRect object will hold the currently selected ActiveX object's size and position.

2. Load the CntrItem.cpp file, and add the following line to the item class's constructor:

```
m_itemRect.SetRect(20, 20, 150, 150);
```

This line initializes the ActiveX object's starting position and size.

3. In the OnGetItemPosition() function, replace the line rPosition.SetRect(10, 10, 210, 210) with the following:

```
rPosition = m_itemRect;
```

This line sets rPosition to the ActiveX object's current rectangle.

4. In the OnChangeItemPosition() function, add the lines shown in listing 26.9 right after the TODO: update any cache you may have of the item's rectangle/extent comment:

### Listing 26.9 lst26\_09.cpp—Line for the OnChangeItemPosition() Function

```
// Get a pointer to the document.
CActiveXCont1Doc* pDoc = GetDocument();

// Redraw all views of this document.
pDoc->UpdateAllViews(NULL);
```

```
// Save the item's new position.
m_itemRect = rectPos;

// Tell MFC that the document needs to be saved.
pDoc->SetModifiedFlag();
```

---

These lines update all views of the document, save the ActiveX object's new position in its `m_itemRect` data member, and tell the document it's been modified (and so needs to be saved).

**5.** Add the following line to the `serialize()` function, right after the `TODO`: add storing code here comment:

```
ar << m_itemRect;
```

This line ensures that the ActiveX object's current position is saved when the document containing the object is saved.

**6.** Add the following line to the `serialize()` function, right after the `TODO`: add loading code here comment:

```
ar >> m_itemRect;
```

This line loads the ActiveX object's last position when the document containing the object is opened.

**7.** Use ClassWizard to add the `OnLButtonDown()` member function to the `CActiveXContlView` class, as shown in Figure 26.9.

**8.** Click the Edit Code button, and add the lines shown in Listing 26.9 to the `OnLButtonDown()` function, right after the `TODO`: Add your message handler code here and/or call default comment. You'll study these lines later in the chapter, in the section entitled "Exploring the `OnLButtonDown()` Function."



**FIG. 26.9** The ClassWizard window should look like this after you've added `OnLButtonDown()`.

---

#### **Listing 26.10** `lst26_10.cpp`—Code for the `OnLButtonDown()` Function

---

```
// Get the item located at the mouse click.
CActiveXContlCntrItem* pHitItem = GetHitItem(point);

// Set the clicked item as selected.
SetSelectedItem(pHitItem);

// If an item was selected...
```

```

if (pHitItem != NULL)
{
    // Create and initialize a rect tracker.
    CRectTracker rectTracker;
    InitTracker(&rectTracker, pHitItem);

    // Update the window's client area.
    UpdateWindow();

    // Enable the user to manipulate the item's rectangle.
    if (rectTracker.Track(this, point))
    {
        // Redraw the window's display.
        Invalidate();

        // Record item's new position.
        pHitItem->m_itemRect = rectTracker.m_rect;

        // Tell MFC that the document needs to be saved.
        CActiveXCont1Doc* pDoc = GetDocument();
        pDoc->SetModifiedFlag();
    }
}

```

---

**9.** Use ClassWizard to add the `OnLButtonDblClk()` member function to the `CActiveXCont1View` class, as shown in Figure 26.10.



**FIG. 26.10** Here, ClassWizard is adding the `OnLButtonDblClk()` message response function.

**10.** Click the Edit Code button, and add the lines shown in Listing 26.11 to the `OnLButtonDblClk()` function, right after the `TODO: Add your message handler code here and/or call default comment`. You'll study these lines later in the chapter, in the section "Exploring the `OnLButtonDblClk()` Function."

---

**Listing 26.11** `lst26_11.cpp`—Code for the `OnLButtonDblClk()` Function

---

```

// Select the clicked item.
OnLButtonDown(nFlags, point);

// If an item was selected...
if (m_pSelection != NULL)
{
    // Get the state of the Ctrl key.
    SHORT keyState = GetKeyState(VK_CONTROL);

```

```

LONG verb;

// If the Ctrl key was pressed,
// the user wants to open the item.
if (keyState < 0)
    verb = OLEIVERB_OPEN;

// Else the user wants to perform the
// item's primary command.
else
    verb = OLEIVERB_PRIMARY;

// Perform the selected action.
m_pSelection->DoVerb(verb, this);
}

```

---

**11.** Use ClassWizard to add the `OnSetCursor()` member function to the `CActiveXControlView` class, as shown in Figure 26.11.



**FIG. 26.11** Here, ClassWizard is adding the `OnSetCursor()` message response function.

**12.** Click the Edit Code button, and add the lines shown in Listing 26.12 to the `OnSetCursor()` function, right after the `TODO`: Add your message handler code here and/or call default comment. You'll study these lines later in the chapter, in the section entitled "Exploring the `OnSetCursor()` Function."

---

**Listing 26.12** `lst26_12.cpp`—Code for the `OnSetCursor()` Function

---

```

// If there's a selected item in this window...
if ((pWnd == this) && (m_pSelection != NULL))
{
    // Create and initialize a rect tracker.
    CRectTracker rectTracker;
    InitTracker(&rectTracker, m_pSelection);

    // If the cursor is over an item's rectangle,
    // enable the tracker to set the mouse cursor.
    BOOL trackerSetCursor =
        rectTracker.SetCursor(this, nHitTest);

    // If the tracker set the cursor,
    // return from the function.
    if (trackerSetCursor)
        return TRUE;
}

```

```
// If the tracker didn't set the cursor,  
// let the window do it.
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)

**13.** Add the functions shown in Listing 26.13 to the end of the `ActiveXCont1View.cpp` file. These new functions initialize a `CRectTracker` object, determine whether the user clicked an embedded or linked item, and get a pointer to the selected item. You'll study these functions in detail later in this chapter, in the sections "Exploring the `InitTracker()` Function," "Exploring the `GetHitItem()` Function," and "Exploring the `SetSelectedItem()` Function."

#### Listing 26.13 `lst26_13.cpp`—New Functions for the View Class

```

void CActiveXCont1View::InitTracker(CRectTracker* pRectTracker,
    CActiveXCont1CntrItem* pItem)
{
    // Get the size of the item.
    pRectTracker->m_rect = pItem->m_itemRect;

    // If the item is selected, draw the resize handles.
    if (pItem == m_pSelection)
        pRectTracker->m_nStyle |= CRectTracker::resizeInside;

    // If the item is linked, surround the
    // item with a dotted line.
    if (pItem->GetType() == OT_LINK)
        pRectTracker->m_nStyle |= CRectTracker::dottedLine;

    // Else if the item is embedded, surround
    // the item with a solid line.
    else
        pRectTracker->m_nStyle |= CRectTracker::solidLine;

    // If the item is being in-place edited, draw
    // a cross-hatch pattern over the item.
    UINT itemState = pItem->GetItemState();
    if ((itemState == ColeClientItem::openState) ||
        (itemState == ColeClientItem::activeUIState))
        pRectTracker->m_nStyle |= CRectTracker::hatchInside;
}

CActiveXCont1CntrItem* CActiveXCont1View::GetHitItem(CPoint point)
{
    CActiveXCont1Doc* pDoc = GetDocument();
    CActiveXCont1CntrItem* pHitItem = NULL;

    // Get the position of the first item.
    POSITION position = pDoc->GetStartPosition();

    // Loop through all the items.
    while (position != NULL)
    {
        // Get a pointer to the next item.
    }
  
```

```

        CActiveXContlCntrItem* pItem =
            (CActiveXContlCntrItem*)pDoc->GetNextItem(position);

        // If the current item contains the clicked point,
        // set the pointer to the hit item.
        BOOL hit = pItem->m_itemRect.PtInRect(point);
        if (hit)
            pHitItem = pItem;
    }

    return pHitItem;
}

void CActiveXContlView::SetSelectedItem(CActiveXContlCntrItem* pItem)
{
    // Get a pointer to the document.
    CActiveXContlDoc* pDoc = GetDocument();

    // If user clicked on an empty part of the window,
    // close the currently active in-place item.
    if (pItem == NULL || m_pSelection != pItem)
    {
        // Get a pointer to the item that's
        // currently being edited in-place.
        COleClientItem* pActiveItem =
            pDoc->GetInPlaceActiveItem(this);

        // Close the in-place item.
        if (pActiveItem != NULL && pActiveItem != pItem)
            pActiveItem->Close();
    }

    // Set the currently selected item.
    m_pSelection = pItem;

    // Redraw the window.
    Invalidate();
}

```

---

**14.** Add the function declarations shown in Listing 26.14 to the `ActiveXContlView.h` header file. Place the lines in the Implementation section, right after the protected keyword.

---

**Listing 26.14** `lst26_14.cpp`—New Function Declarations

---

```

void InitTracker(CRectTracker* pRectTracker,
    CActiveXContlCntrItem* pItem);
CActiveXContlCntrItem* GetHitItem(CPoint point);
void SetSelectedItem(CActiveXContlCntrItem* pItem);

```

---

**15.** In the `OnDraw()` function, replace all the lines following the `TODO`: remove this code when final draw code is complete comment with the lines shown in Listing 26.15. You'll examine these lines in "Examining the `OnDraw()` Function."

---

**Listing 26.15** `lst26_15.cpp`—Lines for the `OnDraw()` Function

---

```

// Get the position of the first item.
POSITION position = pDoc->GetStartPosition();

```

```
// Loop through all items.
while (position != NULL)
{
    // Get a pointer to the next item.
    CActiveXContlCntrItem* pItem =
        (CActiveXContlCntrItem*)pDoc->GetNextItem(position);

    // Draw the item.
    pItem->Draw(pDC, pItem->m_itemRect);

    // Draw the appropriate tracker rectangle on the item.
    CRectTracker rectTracker;
    InitTracker(&rectTracker, pItem);
    rectTracker.Draw(pDC);
}
```

---

You've now created part 2 of the ActiveXCont1 application. To compile the program, select Developer Studio's Build, Build command. Then, select Build, Execute to run the program.

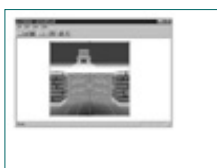
## Selecting ActiveX Objects

When you run the new version of ActiveXCont1, you see the application's main window. Go ahead and link a copy of the Aztec.bmp file into the application's currently open document. When you do, the application's window looks something like Figure 26.12. As you can see, because the bitmap was just added to the document, it is the currently selected item. This means that the bitmap item is surrounded by a selection rectangle that includes handles for moving and resizing the item.



**FIG. 26.12** Here's ActiveXCont1's windows after linking the Aztec.bmp file.

To move the bitmap, just drag it with the mouse pointer. Notice how the mouse pointer changes when it's over the item. To resize the item, use the mouse pointer to manipulate the item's sizing handles. Again, when the mouse pointer is over one of the handles, its form changes to indicate the operation you're about to perform on the item. Figure 26.13 shows the bitmap item after it's been moved and resized.



**FIG. 26.13** You can use your mouse to move and resize the bitmap item.

To deselect the bitmap object, just click inside the window. When you do, the sizing handles disappear. The dotted outline, however, remains. The dotted outline indicates that the item is linked. Had you embedded the item, it would have a solid outline. Reselecting the bitmap object is just a matter of clicking it with the mouse. To edit the bitmap with its server application, double-click the item. Because the item is linked, the server application runs in its own window, as shown in Figure 26.14.



**FIG. 26.14** In most cases, Microsoft Paint is the server application for bitmaps.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Understanding Part 2 of ActiveXCont1

Although AppWizard created a fairly powerful skeleton program for you, when you added mouse item-selection, you had quite a bit of work to do. In this section, you'll explore the code lines that implement this important ActiveX feature.

### Exploring Changes to the *ActiveXCont1CntrlItem* Class

As you already know, in the ActiveXCont1 application, the ActiveXCont1CntrlItem class represents ActiveX objects that are linked or embedded into a document. In the original version of the application, these objects were positioned at a default location and given a default size. In the new version of the application, you've taken over control of an item's position and size. First, you created a data member that you can use to store the item's current location and size:

```
CRect m_itemRect;
```

In the ActiveXCont1CntrlItem class's constructor, the program initializes the m\_itemRect data member like this:

```
m_itemRect.SetRect(20, 20, 150, 150);
```

To change the initial size of embedded or linked items, you only need to change the values to which the CRect object is initialized.

When MFC needs to know the position of an item, it calls the item's OnGetItemPosition() member function, whose single parameter is a reference to a CRect object named rPosition. All your program needs to do is set this CRect object to the rectangle object holding the item's current size and position, like this:

```
rPosition = m_itemRect;
```

When the user decides to move the item, MFC calls the item's OnChangeItemPosition() member function, giving your program a chance to redraw all views displaying the item and to record the new position into the m\_itemRect data member. If you have only a single view of the item, you can probably get away without redrawing the views, but it's always a good idea to follow MFC's rules. To redraw the views, you first get a pointer to the document object and then call the UpdateAllViews() member function through

that pointer, like this:

```
CActiveXContlDoc* pDoc = GetDocument();  
pDoc->UpdateAllViews(NULL);
```

Storing the item's new position is as easy as copying the rectPos parameter to the m\_itemRect data member:

```
m_itemRect = rectPos;
```

You'll also want to tell MFC that the document has been modified. Then, if the user tries to exit the application before saving the document, MFC displays a suitable warning. Calling the document object's SetModifiedFlag() member function takes care of this problem:

```
pDoc->SetModifiedFlag();
```

Finally, you want the CActiveXContlCntrlItem class to save the location of its items when it saves its document. You do this by adding lines to the class's Serialize() member function. This line saves the value of the CRect object:

```
ar << m_itemRect;
```

This line, on the other hand, reads the CRect object back in from the file:

```
ar >> m_itemRect;
```

## Exploring the *OnLButtonDown()* Function

When the user wants to select or deselect an item, he clicks the item or the window with his mouse. This means that the OnLButtonDown() message-response function, which MFC calls in response to mouse clicks, is the perfect place to implement mouse selection of ActiveX objects. In the CActiveXContlView class, the OnLButtonDown() function first determines whether an ActiveX item has been clicked:

```
CActiveXContlCntrlItem* pHitItem = GetHitItem(point);
```

GetHitItem() is a local function that you added to the program. You'll examine that function in detail soon, in the section entitled "Exploring the GetHitItem() Function." For now, just know that the function returns a pointer to the item at the location of the mouse click or returns NULL if no item was clicked. The location of the mouse click is conveniently passed to OnLButtonDown() in the point parameter.

After getting a pointer to the clicked item (if any), the program calls SetSelectedItem() to determine whether the mouse click was meant to select or deselect an item:

```
SetSelectedItem(pHitItem);
```

Again, SetSelectedItem() is a local function that you added to the program. You'll examine this function in detail in the section entitled "Exploring the SetSelectedItem() Function."

Next, OnLButtonDown() checks whether an item was actually selected:

```
if (pHitItem != NULL)
```

If so, the program creates and initializes a CRectTracker object:

```
CRectTracker rectTracker;  
InitTracker(&rectTracker, pHitItem);
```

A CRectTracker object is responsible for displaying a selection rectangle around an item, including displaying the item's selection handles. This process is handled in the InitTracker() function, which is a member function you added to the class. You'll examine this function in the section called "Exploring the InitTracker() Function." All you need to know now is that InitTracker() sets up the CRectTracker object as appropriate for the type of item that's been selected.

After setting up the CRectTracker object, the program calls the CRectTracker's Track() member function, which enables the user to manipulate the item:

```
if (rectTracker.Track(this, point))
```

The Track() function returns TRUE if the user completes the item manipulation. In this case, the program must redraw the item in its new location or size and store the item's new rectangle:

```
Invalidate();  
pHitItem->m_itemRect = rectTracker.m_rect;
```

Finally, the program notifies MFC that the document has been changed and needs to be saved:

```
CActiveXCont1Doc* pDoc = GetDocument();  
pDoc->SetModifiedFlag();
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Exploring the *OnLButtonDbtClk()* Function

When the user wants to edit an item, she double-clicks it. So, item-editing program lines should be placed in the view class's *OnLButtonDbtClk()* member function, which MFC calls when the user double-clicks in a window.

In the *CActiveXContlView* class, the first thing *OnLButtonDbtClk()* does is call *OnLButtonDown()* to handle the item-selection process:

```
OnLButtonDown(nFlags, point);
```

The function then checks that an item has been selected:

```
if (m_pSelection != NULL)
```

If so, the program gets the state of the keyboard's Ctrl key:

```
SHORT keyState = GetKeyState(VK_CONTROL);
```

If the Ctrl key were pressed (*keyState < 0*) during the double-click, the user wants to open the selected item. The predefined verb (as ActiveX item commands are called) for this action is *OLEIVERB\_OPEN*:

```
if (keyState < 0)
    verb = OLEIVERB_OPEN;
```

If the Ctrl key weren't pressed, the user is requesting that the item's primary command, which is represented by the predefined *OLEIVERB\_PRIMARY* value, be executed:

```
else
    verb = OLEIVERB_PRIMARY;
```

To execute the selected item verb, the program calls the selected item's *DoVerb()* member function, like this:

```
m_pSelection->DoVerb(verb, this);
```

## Exploring the *OnSetCursor()* Function

Whenever the user places the mouse pointer over a selected linked or embedded item, the pointer should change to show the action that the user can currently



perform with the mouse. This happens in the view class's OnSetCursor() function. This function first checks whether there's a selected item in the window:

```
if ((pWnd == this) && (m_pSelection != NULL))
```

If there's no selected item, OnSetCursor() just handles the mouse cursor the normal way by calling the base class's OnSetCursor(). If there is a selected item, the function creates and initializes a CRectTracker object for the currently selected item:

```
CRectTracker rectTracker;  
InitTracker(&rectTracker, m_pSelection);
```

The function then calls the CRectTracker object's SetCursor() member function to set the mouse pointer to the appropriate shape for the part of the tracker rectangle the pointer is over:

```
BOOL trackerSetCursor =  
    rectTracker.SetCursor(this, nHitTest);
```

The tracker's SetCursor() function returns TRUE if it handles the pointer and FALSE if it doesn't. The function uses the returned value, stored in trackerSetCursor, to determine whether to return the following from the function:

```
if (trackerSetCursor)  
    return TRUE;
```

## Exploring the *OnDraw()* Function

As you know from many previous MFC programs, the OnDraw() function is responsible for drawing a view window's display. This is true for ActiveX applications, as well, except that ActiveX containers must draw both native data and any linked or embedded items. Because the current document can contain more than one linked or embedded item, the OnDraw() function must locate and draw all the items. To do this, the function first gets the position of the first item by calling the document object's GetStartPosition() member function:

```
POSITION position = pDoc->GetStartPosition();
```

If this function returns NULL, there is no first object to find. However, if it returns the position of an object, OnDraw() begins a while loop that'll iterate through all the items in the document:

```
while (position != NULL)
```

Inside the loop, the program gets a pointer to the item located at the position stored in the position variable:

```
CActiveXContlCntrItem* pItem =  
    (CActiveXContlCntrItem*)pDoc->GetNextItem(position);
```

OnDraw() can then draw the item by calling the item's Draw() function:

```
pItem->Draw(pDC, pItem->m_itemRect);
```

This function requires two parameters, which are a pointer to the device context and the CRect object, holding the item's size and position.

Finally, because an item drawn might be selected, the appropriate CRectTracker rectangle must be drawn. OnDraw() handles this important task like this:

```
CRectTracker rectTracker;  
InitTracker(&rectTracker, pItem);  
rectTracker.Draw(pDC);
```

Although not done in ActiveXCont1, any native data (data that was created by the container application, such as the text is a word processor's document), should also be drawn in OnDraw(), just as it normally would be drawn in a non-ActiveX application.

## Exploring the *InitTracker()* Function

Before a CRectTracker object can be used, it must be initialized. How this is done depends upon the type of item with which the tracker is being used. In the ActiveXCont1 application, the InitTracker() function takes care of these details. First, InitTracker() sets the size of the tracker rectangle to that of the item, like this:

```
pRectTracker->m_rect = pItem->m_itemRect;
```

Then, if the item is selected, the tracker must display resize handles, which is done by adding the CRectTracker::resizeInside style to the tracker's m\_nStyle style data member:

```
if (pItem == m_pSelection)  
    pRectTracker->m_nStyle |= CRectTracker::resizeInside;
```

If the item is linked into the document, it must have a dotted outline, which is represented by the tracker style CRectTracker::dottedLine:

```
if (pItem->GetType() == OT_LINK)  
    pRectTracker->m_nStyle |= CRectTracker::dottedLine;
```

If the item is embedded rather than linked, it needs a solid outline:

```
else  
    pRectTracker->m_nStyle |= CRectTracker::solidLine;
```

A little trickier is the process of determining whether or not the object is in its open and active state. In such a case, the item needs to be overlaid with a hatch pattern. To do this, InitTracker() first retrieves the item's state by calling the item's GetItemState() member function:

```
UINT itemState = pItem->GetItemState();
```

The program can then compare the state to the flags that indicate whether the item is active. If it is, the CRectTracker::hatchInside style is added to the tracker rectangle:

```
if ((itemState == COleClientItem::openState) ||  
    (itemState == COleClientItem::activeUIState))  
    pRectTracker->m_nStyle |= CRectTracker::hatchInside;
```

When the tracker rectangle is drawn, the styles that were set by InitTracker() will control the tracker rectangle's appearance.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Exploring the *GetHitItem()* Function

When the user clicks in the application's window, the application must determine whether an embedded or linked item has been selected. This means comparing the coordinates of the mouse click with the coordinates of each item in the document. This task is handled by the *GetHitItem()* function. The function first gets a pointer to the document object and sets *pHitItem*, which will hold a pointer to the selected item (if any), to NULL:

```
CActiveXCont1Doc* pDoc = GetDocument();
CActiveXCont1CntrItem* pHitItem = NULL;
```

Next, the function gets the position of the first linked or embedded item:

```
POSITION position = pDoc->GetStartPosition();
```

Much like *OnDraw()*, *GetHitItem()* uses a while loop to cycle through all the items in the document. Inside this loop, the function first gets a pointer to the item at the current position:

```
CActiveXCont1CntrItem* pItem =
    (CActiveXCont1CntrItem*)pDoc->GetNextItem(position);
```

The program then checks whether the clicked point is contained in the item's rectangle:

```
BOOL hit = pItem->m_itemRect.PtInRect(point);
```

If *PtInRect()* returns TRUE, the function has located the item to be selected:

```
if (hit)
    pHitItem = pItem;
```

Finally, *pHitItem* (which can be NULL, indicating that no item was clicked) is returned from the function:

```
return pHitItem;
```

## Exploring the *SetSelectedItem()* Function

When the user clicks in the application's window, the program must determine whether the user is selecting a new item or deselecting an already selected item. In

some cases, the user might be doing both, such as when he clicks on a new item when another item is already activated in-place. The selection and deselection tasks are handled in the `SetSelectedItem()` function. First, this function gets a pointer to the document object:

```
CActiveXCont1Doc* pDoc = GetDocument();
```

The function then checks whether the user clicked on an empty part of the window (`pItem == NULL`) or is trying to select a new item when a previous item is still activated (`m_pSelection != pItem`):

```
if (pItem == NULL || m_pSelection != pItem)
```

If neither of these cases is true, the function simply sets `m_pSelection` to the newly selected item:

```
m_pSelection = pItem;
```

However, if the one of the two previous tests is true, there's a previously activated item that needs to be deactivated. Getting a pointer to the currently active item is a simple matter of calling the document object's `GetInPlaceActiveItem()` member function:

```
COleClientItem* pActiveItem =  
    pDoc->GetInPlaceActiveItem(this);
```

Finally, if there is an active item, and that item isn't the item being currently selected, the program closes the active item by calling the item's `Close()` member function:

```
if (pActiveItem != NULL && pActiveItem != pItem)  
    pActiveItem->Close();
```

The last step is to redraw the document's display:

```
Invalidate();
```

## Deleting ActiveX Objects

Currently, as soon as you place a linked or embedded item into an `ActiveXCont1` document, you're stuck with it. Short of starting a new document, there's no way to get rid of unwanted items. As you'll see in the following steps, enabling the user to delete items from an ActiveX document is easy to do. Just perform the following steps to add this capability to the `ActiveXCont1` application.



The complete source code and executable file for this version of the `ActiveXCont1` application is located in the `Chap26\ActiveXCont1, Part 3` folder on this book's CD-ROM.

---

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

1. Use Developer Studio's menu editor to add a Delete menu item to the application's Edit menu (found in the IDR\_MAINFRAME menu), as shown in Figure 26.15. Give the new menu item the ID\_EDIT\_CLEAR menu ID.



**FIG. 26.15** Give the new Delete menu the ID\_EDIT\_CLEAR menu ID.

2. Drag the new menu item so that it's positioned just below the Paste Special item, as shown in Figure 26.16.



**FIG. 26.16** The Delete menu should be positioned as shown here.

3. Bring up ClassWizard by pressing Ctrl-W. When the ClassWizard window appears, select CActiveXCont1View in the Class Name box.
4. Use ClassWizard to add to the view class both COMMAND and UPDATE\_COMMAND\_UI message-response functions for the Delete menu item, accepting the suggested function names, as shown in Figure 26.17.



**FIG. 26.17** To handle the new Delete menu item, you need COMMAND and UPDATE\_COMMAND\_UI functions.

5. Click the Edit Code button, and add the lines shown in Listing 26.16 to the OnEditClear() function. You'll examine these lines later in this chapter, in the section entitled "Exploring the OnEditClear() Function."

#### Listing 26.16 lst26\_16.cpp—Lines for the OnEditClear() Function

```

if (m_pSelection != NULL)
{
    // Delete the selected item.
    m_pSelection->Delete();
    m_pSelection = NULL;

    // Update all view windows associated
    // with this document.
    CActiveXCont1Doc* pDoc = GetDocument();
    pDoc->UpdateAllViews(NULL);
}

```

---

**6.** Add the lines shown in Listing 26.17 to the OnUpdateEditClear() function.

These lines simply enable or disable the Delete menu item, depending on whether an embedded or linked item is currently selected.

---

**Listing 26.17** lst16\_17.cpp—Lines for the OnUpdateEditClear() Function

---

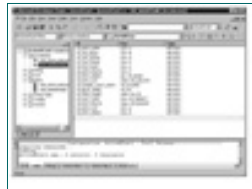
```

BOOL enable = TRUE;
if (m_pSelection == NULL)
    enable = FALSE;
pCmdUI->Enable(enable);

```

---

**7.** Bring up the application's IDR\_MAINFRAME accelerator table, as shown in Figure 26.18.



**FIG. 26.18** To add hotkeys to menu items, you must first display the appropriate accelerator table.

**8.** Add an accelerator key for the Delete command, as shown in Figure 26.19. Be sure to select the ID\_EDIT\_CLEAR and VK\_DELETE IDs and to turn off all options in the Modifiers box.



**FIG. 26.19** The VK\_DELETE virtual-key ID is associated with the keyboard's Delete key.

You've now created part 3 of the ActiveXCont1 application. To compile the program, choose Developer Studio's Build, Build command. Then, select Build, Execute to run the program.



## Using the Third Version of ActiveXCont1

Testing the ActiveXCont1 application's new capabilities won't take more than two minutes. When you run the program, link or embed a new item. Then, select the new item and press your keyboard's Delete key. *Presto!* The item is deleted from the document. You'll get the same results if you use the application's Edit, Delete command instead of pressing the Delete key.

### Exploring the *OnEditClear()* Function

At this point, you should easily understand most of the new program lines and functions that you just added to the ActiveXCont1 application. However, the OnEditClear() function might require a little explanation.

When the user selects an embedded or linked item and clicks the Edit, Delete command or presses the Delete key, OnEditClear() springs into action. The function first checks whether an item is selected:

```
if (m_pSelection != NULL)
```

If so, deleting the item is as simple as calling the item's Delete() member function and setting the item's pointer to NULL:

```
m_pSelection->Delete();  
m_pSelection = NULL;
```

The final task is to update all views that might be connected to the document that contained the deleted item:

```
CActiveXCont1Doc* pDoc = GetDocument();  
pDoc->UpdateAllViews(NULL);
```

If no item is selected, OnEditClear() does nothing but return.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Chapter 27

# Creating an ActiveX Server Application

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

- Learn how to use AppWizard to create a skeleton ActiveX server application
- Discover how the server classes and functions created by AppWizard work
- Learn how to make the server application supply the type of item specific to your application
- See how to use mapping modes and extents so that a linked or embedded item looks the same on the screen when it's activated or deactivated
- Explore how to create menus for your server application

In the previous chapter, "Creating an ActiveX Container Application," you created an ActiveX container application, which can link or embed ActiveX items. A container isn't much good, however, unless there is an ActiveX server registered with the system. It is, after all, the server that provides the linkable and embeddable items to the container. When you tested your container application, you used Microsoft Paint, one of the ActiveX servers that comes with Windows 95. In this chapter, you learn to create your own servers.

## Creating the Basic ActiveX Server Application

Just as AppWizard can create a functional container application in a few steps, so too can it create a functional server application. The skeleton application created by AppWizard features all the basic functionality of an ActiveX server application, including the capability to embed, link, and edit ActiveX objects in-place in a container application's window. Later in this section, you'll see how to use the server and container applications together. But first you must create the basic server application, which you can do by completing the following steps.

1. Start a new AppWizard project workspace called ActiveXServ.
2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 27.1.

- Step 1: Single document
- Step 2: Default settings
- Step 3: Select Full Server
- Step 4: Default settings
- Step 5: Default settings
- Step 6: Default settings



**FIG. 27.1** These are the AppWizard settings for the ActiveXServ project.

---

**Note:**

The complete source code and executable file for this version of the ActiveXServ application is located in the Chap27\ActiveXServ, Part 1 folder on this book's CD-ROM.

---



You've now created the basic ActiveXServ application. To compile the program, select Developer Studio's Build, Build command. Then, select Build, Execute to run the program. When you do, you see the window shown in Figure 27.2. Thanks to AppWizard and MFC, the program is already a powerful server application that can provide linked and embedded items to a container application.

---

**Note:**

Running the ActiveXServ application is all you have to do to register the application with the system as an ActiveX server.

---



**FIG. 27.2** The ActiveXServ application's mainwindow looks like this.

To see that the new server application actually works, close ActiveXServ and then run the completed ActiveXCont1 application that you created in Chapter 26, "Creating an ActiveX Container Application." Select the Edit, Insert New Object command. The Insert Object dialog box appears. In the Object Type box, you'll see a new object called Active Document (see Figure 27.3). This is the type of item that is supplied by the new ActiveXServ server application.



**FIG. 27.3** The Insert Object dialog box lists the item that ActiveXServ supplies.

You can actually embed an Active Document item into the container application. Just double-click Active Document in the Object Type box. When you do, the ActiveXServ application supplies a new Active Document item and opens the item for editing, changing the toolbar and menus to that of the server application (see Figure 27.4). To prove that the menus and toolbar really come from the server, click the question mark button on the toolbar. The ActiveXServ server application's About dialog box appears, as shown in Figure 27.5.



**FIG. 27.4** ActiveXServ supports in-place editing of Active Document items.



**FIG. 27.5** The About dialog box proves that the server application is running in-place.

When you're finished experimenting with ActiveXCont1 and its new Active Document item, close the application.

## Understanding the ActiveXServ Skeleton Application

As you saw in the section "Creating the Basic ActiveX Server Application," AppWizard has already supplied ActiveXServ with basic server functionality. To do this, AppWizard generated a lot of source code for the ActiveXServ project. In this section, you'll examine much of that code to see how the server application works.

---

### Note:

As you look through MFC's ActiveX code, you'll notice that the word OLE comes up a lot. This is because MFC's classes and member functions were named before OLE was changed to ActiveX.

---

## Exploring the CActiveXServApp Class

To understand the source code, the first place to look is in InitInstance(), where the application performs its initialization at startup. That function, which is defined as part of the CActiveXServApp class, performs its normal initialization, as well as some special initialization for ActiveX. To initialize the ActiveX

libraries, InitInstance() calls AfxOleInit(), as shown in Listing 27.1:

**Listing 27.1 lst27\_01.cpp—Calling AfxOleInit()**

---

```
if ( !AfxOleInit() )
{
    AfxMessageBox( IDP_OLE_INIT_FAILED ) ;
    return FALSE ;
}
```

---

The InitInstance() function also calls the AfxEnableControlContainer() global function, which enables the application to act as a container for ActiveX controls. That function call looks like this:

```
AfxEnableControlContainer( ) ;
```

When creating the template for the application's document class, InitInstance() must call the CSingleDocTemplate class's SetServerInfo() member function to supply information about the server:

```
pDocTemplate-> SetServerInfo(
    IDR_SRVR_EMBEDDED, IDR_SRVR_INPLACE,
    RUNTIME_CLASS( CInPlaceFrame ) ) ;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Here, the IDR\_SRVR\_EMBEDDED is the ID that represents the server application's menu and accelerator resources. These resources are used when the server application has been run in its own window from the container application. The IDR\_SRVR\_INPLACE ID represents the menus, toolbar, and accelerators needed by the container when an Active Document item is being edited in-place. The CInPlaceFrame class represents the window that is supplied by the server to the container application when an Active Document item is activated in-place.

Next, InitInstance() calls the server object's ConnectTemplate() member function to register the document with the system:

```
m_server.ConnectTemplate(clsid, pDocTemplate, TRUE);
```

The clsid argument is the GUID for the server that was created automatically when the application was generated. (**GUID** stands for **Globally Unique Identifier**, and each application receives its own so that the application can be registered in the system.) This GUID will be different for every Visual C++ server project. For the version of ActiveXServ that's on this book's CD-ROM, clsid is defined like this:

```
static const CLSID clsid =
{ 0xb28d4202, 0x6a0f, 0x11d0,
  { 0x84, 0x7f, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };
```

The m\_server data member, an object of the COleTemplateServer class, is defined in the CActiveXServApp class's header file.

Now, if the server application is being run in place or as an automation server (you'll learn about automation in Chapter 28, "Discovering OLE Automation"), the application shouldn't display its own window. InitInstance() returns before creating a window if this is the case, thanks to the program lines shown in Listing 27.2:

### Listing 27.2 lst27\_02.cpp—Code that Handles In-place or Automation Servers

```
// Check to see if launched as OLE server
if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
{
    // Register all OLE server (factories) as running. This enables the
    // OLE libraries to create objects from other applications.
    COleTemplateServer::RegisterAll();

    // Application was run with /Embedding or /Automation. Don't show the
    // main window in this case.
    return TRUE;
}
```

The call to COleTemplateServer::RegisterAll() in Listing 27.2 performs more required OLE initialization for MFC.

Finally, before creating and displaying the application's window, the following line registers the application as an ActiveX server:

```
m_server.UpdateRegistry(OAT_INPLACE_SERVER);
```

## Exploring the *CActiveXServSrvrItem* Class

Like everything else in MFC, an Active Document item (which is the type of item that ActiveXServ can embed into a container application's document) is represented by an instance of a class. MFC provides the *COleServerItem* class to represent these types of objects. In the ActiveXServ application's source code, the *CActiveXServSrvrItem* class is derived from *COleServerItem* and represents embedded objects for this specific application.

The *CActiveXServSrvrItem* class, as provided by AppWizard, contains a number of member functions important to the process of embedding or linking Active Document items. First, the *Serialize()* function shown in Listing 27.3 doesn't serialize the item to disk, as usually happens with other *Serialize()* functions. Instead, the function handles embedded or linked items that are being copied to the Clipboard. Because an embedded item represents an entire document, AppWizard can provide default code for serializing the item. For linked items, however, you have to supply the code that serializes only the portion of the document that the user chose to link. For example, the user might have linked a paragraph from a word processor's document. Because the ActiveXServ application cannot supply partial documents for linking, you don't need to modify *Serialize()*.

### Listing 27.3 *lst27\_03.cpp*—The *Serialize()* Member Function

---

```
void CActiveXServSrvrItem::Serialize(CArchive& ar)
{
    // CActiveXServSrvrItem::Serialize will be
    // called by the framework if
    // the item is copied to the clipboard.
    // This can happen automatically
    // through the OLE callback OnGetClipboardData.
    // A good default for
    // the embedded item is simply to delegate
    // to the document's Serialize
    // function. If you support links, then
    // you will want to serialize
    // just a portion of the document.

    if (!IsLinkedItem())
    {
        CActiveXServDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->Serialize(ar);
    }
}
```

---

Next is the *OnGetExtent()* member function, which is shown in Listing 27.4. *OnGetExtent()* returns the item's extent (size). In most cases, the extent will be the size of the entire item as it normally appears on the screen. However, you can change the extent to accommodate things like drawing thumbnail or iconic images of the item. As you can see, the AppWizard-generated version of this function uses a hard-coded extent of 3000 × 3000, measured in HIMETRIC units.

### Listing 27.4 *lst27\_04.cpp*—The *OnGetExtent()* Member Function

---

```
BOOL CActiveXServSrvrItem::OnGetExtent(DVASPECT dwDrawAspect,
    CSize& rSize)
{
    // Most applications, like this one, only handle
    // drawing the content
    // aspect of the item. If you wish to support other aspects, such
    // as DVASPECT_THUMBNAIL (by overriding OnDrawEx), then this
    // implementation of OnGetExtent should be modified to handle the
    // additional aspect(s).
```

```
if (dwDrawAspect != DVASPECT_CONTENT)
    return COleServerItem::OnGetExtent(dwDrawAspect, rSize);

// CActiveXServSrvrItem::OnGetExtent is called to get the extent in
// HIMETRIC units of the entire item. The default implementation
// here simply returns a hard-coded number of units.

CActiveXServDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);

// TODO: replace this arbitrary size

rSize = CSize(3000, 3000);    // 3000 x 3000 HIMETRIC units

return TRUE;
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Finally, the OnDraw() member function, as shown in Listing 27.5, is responsible for displaying the item in the container when the item is not active. The default version of this function supplied by AppWizard simply sets up the mapping mode and window extent. You must add your own code to actually draw the item in the container application's window, which you'll do later in this chapter, in the section entitled "Completing the ActiveXServ Server Application."

#### Listing 27.5 lst27\_04.cpp—The OnDraw() Member Function

```

BOOL CActiveXServSrvrItem::OnDraw(CDC* pDC, CSize& rSize)
{
    CActiveXServDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: set mapping mode and extent
    // (The extent is usually the same as the
    // size returned from OnGetExtent)
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowOrg(0,0);
    pDC->SetWindowExt(3000, 3000);

    // TODO: add drawing code here. Optionally,
    // fill in the HIMETRIC extent.
    // All drawing takes place in the metafile device context (pDC).

    return TRUE;
}
  
```

### Exploring the CActiveXServView Class

The application's view class doesn't look much different from other AppWizard-generated view classes. One big difference is that the class is now able to draw the application's document in its own window, as well as in a container application's window when activated in-place. All of this activity, however, happens in the background. There's little extra you need to do besides implement OnDraw(), as you've always done for any view class.

About the only thing you'll see in the view class that's directly related to being an ActiveX server is the OnCancelEditSrvr() member function (see Listing 27.6), which responds when the user presses the Esc key when editing an Active Document item in-place in a container application. This action cancels the editing and closes the item. Later in this chapter, in the section entitled "Completing the ActiveXServ Server Application," you'll add program lines to the view class that draw an Active Document and that enable the user to edit the document.

#### Listing 27.6 lst27\_06.cpp—The OnCancelEditSrvr() Member Function

```

void CActiveXServView::OnCancelEditSrvr()
{
  
```

```
        GetDocument() -> OnDeactivateUI( FALSE );  
    }
```

---

## Exploring the *CActiveXServDoc* Class

One important difference between the document classes you're used to working with and the *CActiveXServDoc* class is that the latter is derived from MFC's *COleServerDoc* class, rather than from *CDocument*. Much of the ActiveX server functionality is built into the *COleServerDoc* class.

The class's constructor, as shown in Listing 27.7, is responsible for enabling compound files, which are very important to documents that contain linked or embedded items in addition to native data. You'll add additional construction code to the constructor when you complete the ActiveXServ application.

### Listing 27.7 *lst27\_07.cpp*—The *CActiveXServSrvrDoc* Class's Constructor

---

```
CActiveXServDoc::CActiveXServDoc()  
{  
    // Use OLE compound files  
    EnableCompoundFile();  
  
    // TODO: add one-time construction code here  
}
```

---

When the container application activates the server to edit a new linked or embedded item, MFC calls the document class's *OnGetEmbeddedItem()* member function. As you can see in Listing 27.8, *OnGetEmbeddedItem()* creates a new item of the appropriate type (in the case of the ActiveXServ application, the type is *CActiveXServSrvrItem*) and returns a pointer to the item from the function.

### Listing 27.8 *lst27\_08.cpp*—The *OnGetEmbeddedItem()* Member Function

---

```
COleServerItem* CActiveXServDoc::OnGetEmbeddedItem()  
{  
    // OnGetEmbeddedItem is called by the framework  
    // to get the COleServerItem  
    // that is associated with the document.  
    // It is only called when necessary.  
  
    CActiveXServSrvrItem* pItem = new CActiveXServSrvrItem(this);  
    ASSERT_VALID(pItem);  
    return pItem;  
}
```

---

## Exploring the *CInPlaceFrame* Class

The last class of interest is *CInPlaceFrame*, which represents the window in which an item is edited within a container application. That is, when the user opens an item for in-place editing, MFC creates a *CInPlaceFrame* window and positions the window in the container application's frame window. However, even though the *CInPlaceFrame* window is located in the container's window, the server is responsible for it. The *CInPlaceFrame* window must also manage the server's control bars. This occurs when the class's *OnCreate()* calls the *OnCreateControlBars()* member function, which is shown in Listing 27.9.

### Listing 27.9 *lst27\_09.cpp*—The *OnCreateControlBars()* Member Function

---

```
BOOL CInPlaceFrame::OnCreateControlBars(CFrameWnd* pWndFrame,
```

```

CFrameWnd* pWndDoc)
{
    // Set owner to this window, so messages are
    // delivered to correct app
    m_wndToolBar.SetOwner(this);

    // Create toolbar on client's frame window
    if (!m_wndToolBar.Create(pWndFrame) ||
        !m_wndToolBar.LoadToolBar(IDR_SRVR_INPLACE))
    {
        TRACE0("Failed to create toolbar\n");
        return FALSE;
    }

    // TODO: Remove this if you don't want tool
    // tips or a resizable toolbar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    // TODO: Delete these three lines if you don't want the toolbar to
    // be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    pWndFrame->EnableDocking(CBRS_ALIGN_ANY);
    pWndFrame->DockControlBar(&m_wndToolBar);

    return TRUE;
}

```

---

## Completing the ActiveXServ Server Application

You're now ready to add the application-specific program lines that will complete the ActiveXServ application. These program lines draw the data that represent an Active Document item, as well as enable the user to edit the item, both in the application's main window and as an in-place item in a container application's window. Just complete the following steps to finish the application.

---

### Note:

The complete source code and executable file for this version of the ActiveXServ application is located in the Chap27\ActiveXServ, Part 2 folder on this book's CD-ROM.

---



1. Load the ActiveXServDoc.h header files, and add the following line to the class's Attributes section, right after the public keyword:

```
CString m_string;
```

This line defines the string that holds the data for an ActiveXServ document.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

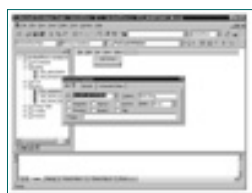
[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

**14.** Add the following line to the top of the ActiveXServView.cpp file, right after the #endif line:

```
#include "stringdlg.h"
```

**15.** Load the IDR\_MAINFRAME menu into Developer Studio's menu editor. Add a Set menu, as shown in Figure 27.11. Give the Set String command an ID of ID\_SET\_SETSTRING.



**FIG. 27.11** Use the menu editor to add the Set menu.

**16.** Add identical Set menus to the IDR\_SRVR\_EMBEDDED and IDR\_SRVR\_INPLACE menus, as shown in Figures 27.12 and 27.13. Adding the Set menu to all three menu resources ensures that the menu will be available both in the application's main window and in a container application's window when an Active Document item is selected or activated.



**FIG. 27.12** Here's the IDR\_SRVR\_EMBEDDED menu.



**FIG. 27.13** Here's the IDR\_SRVR\_INPLACE menu.

**17.** Use ClassWizard to add the OnSetSetstring() message-response function to the view class, as shown in Figure 27.14.



**FIG. 27.14** The OnSetSetstring() message-response function responds to your new

Set String menu command.

**18.** Click the Edit Code button, and add the lines shown in Listing 27.10 to the OnSetSetstring() function:

---

**Listing 27.10** lst27\_10.cpp—Program Lines for the OnSetSetstring() Function

---

```
CStringDlg dlg;

int result = dlg.DoModal();

if (result == IDOK)
{
    // Reset the string's value.
    CActiveXServDoc* pDoc = GetDocument();
    pDoc->m_string = dlg.m_string;

    // Repaint the window.
    Invalidate();

    // Tell the document that it needs to be saved.
    pDoc->SetModifiedFlag();

    // Notify the document that the container
    // needs to be updated.
    pDoc->NotifyChanged();
}
```

---

## Using the Final Version of ActiveXServ

You've now created the final version of the ActiveXServ application. To compile the program, select Developer Studio's Build, Build command. Then, select Build, Execute to run the program. When you do, you see the application's main window. You can use the Set menu to change the value of the displayed string. However, things get really interesting when you run ActiveXServ as a server application.

To run ActiveXServ as a server application, first close the application's window. Then, run the container application you designed in the previous chapter. Finally, use the Edit, Insert Object command to insert an Active Document object into the container window, as shown in Figure 27.15. As you can see, not only does the new object appear in the container application's window, but also the server application's menu and toolbar appear. This enables you to edit the object almost as if you were still running the main ActiveXServ application, which, for all intents and purposes, you are.



**FIG. 27.15** You can run ActiveXServ as an ActiveX server by inserting one of its

objects into a container application.

To see how the in-place editing works, select the Set, Set String command. The Set String dialog box appears (see Figure 27.16). Type a new string into the dialog box and click OK. The new string appears in the container application's window (see Figure 27.17).

When you click outside of the Active Document object, the object is deselected and the container application's menu and toolbar reappear, as shown in Figure 27.18.

And that's all there is to it! There's not much else to discuss about the basic ActiveX server application. The details are described in the steps you used to build the application, as well as in the comments found in the program lines. Take some time now to experiment with the server application, changing program lines to see how they affect the application. That's the best way to learn MFC's many secrets.



**FIG. 27.16** The Set String dialog box enables you to edit the string.



**FIG. 27.17** The container application displays any changes you make to the data string.



**FIG. 27.18** Deactivating an object brings back the container's menus and toolbar.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 28

# Discovering OLE Automation

- Discover how to use AppWizard to create a skeleton OLE automation application
- Explore the automation application's classes and functions created by AppWizard
- Learn how to create the server application's automation interface
- See how to create a client application that can control an automation server

Perhaps one of the most advanced features of OLE, a feature that's just coming into widespread use, is automation. Because OLE automation enables one application to control or use the services of another, it is the crux of component programming and component reusability. For example, using OLE automation, you can obtain spell checker services simply by controlling another application's spell checker. In this way, applications can take advantage of capabilities already present in the user's system, without having to build common functions like spell checking into their own applications. Programmers are now free to concentrate on an application's specifics, rather than wasting their time reinventing the wheel. In this chapter, you learn to take advantage of OLE automation.

## Creating an Automation Server Application

An *OLE automation server* is an application that provides components that other applications can use. For example, a word processor that provides a spell checker component is an automation server if other applications can use the spell checker's services through OLE automation.

If you had to program an OLE automation server from scratch, you'd need a shelfful of manuals and a year or two of intense OLE training. Although OLE is extremely powerful, nobody ever promised that it would be easy to program! Luckily, Microsoft has built most of the OLE capabilities your application is likely to need right into MFC. Because of this, writing an automation server is easier than you might believe. To create an automation server, you must complete the following steps:

1. Use AppWizard to create an OLE automation skeleton application.
2. Complete the stand-alone part of the application, just as you would



with any other MFC application.

3. Create properties and methods to provide an automation interface for the application.

In the following sections, you'll complete the previous steps as you create AutoServer, a basic automation server application. Although AutoServer is a very simple application, it demonstrates how to create a server application's automation interface. You can then apply these basics toward the creation of more sophisticated applications.

## Creating the Basic AutoServer Application

The first step in creating an automation server is to use AppWizard to create a skeleton application that supports OLE automation. This is as easy as selecting a single option in the AppWizard dialog boxes. The second step, after creating the skeleton application, is to complete the program so that it can run as a stand-alone application. Complete the following steps to create the AutoServer stand-alone application. Later, in the section "Adding Properties and Methods to the Automation Server," you'll create the application's OLE automation interface.



The complete source code and executable file for this version of the ActiveXServ application is located on this book's CD-ROM.

1. Start a new AppWizard project workspace called AutoServer.
2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 28.1.

Notice that all you need to do to create a skeleton automation server is to choose the OLE Automation option in the Step 3 dialog box.

Step 1: Single document

Step 2: Default settings

Step 3: Select OLE Automation

Step 4: Default settings

Step 5: Default settings

Step 6: Default settings

3. Load the AutoServerDoc.h header file, and add the following line to the class's Attributes section, right after the public keyword:

```
UINT m_x;
```

This line defines the integer that holds the editable data for an AutoServer document.



**FIG. 28.1** These are the project settings for the AutoServer application.

4. Load the AutoServerDoc.cpp file, and add the following line to the class's constructor, right after the TODO: add one-time construction code here comment:

```
m_x = 20;
```

This line initializes the data to its default value.

5. Add the following line to the Serialize() function, right after the TODO: add storing code here comment:

```
ar << m_x;
```

6. Add the following line to the Serialize() function, right after the TODO: add loading code here comment:

```
ar >> m_x;
```

7. Load the AutoServerView.cpp file, and add the following line to the OnDraw() function, right after the TODO: add draw code for native data here comment:

```
pDC->TextOut(pDoc->m_x, 20, "TEST STRING");
```

This line displays the document's test string, using the X position found in the document class's m\_x data member.

8. Load the IDR\_MAINFRAME menu into Developer Studio's menu editor. Add a Set menu, as shown in Figure 28.2. Give the Set X Position command an ID of ID\_SET\_SETXPOSITION.

9. Use ClassWizard to add the OnSetSetxposition() message response function to the document class, as shown in Figure 28.3.



**FIG. 28.2** Use the menu editor to add the Set menu.



**FIG. 28.3** The OnSetSetxposition() message response function responds to your new Set X Position menu command.

10. Click the Edit Code button, and add the lines shown in Listing 28.1 to the OnSetSetxposition() function:

**Listing 28.1** lst28\_01.cpp—Program Lines for the OnSetSetxposition()

## Function

---

```
if (m_x == 20)
    m_x = 300;
else
    m_x = 20;

SetModifiedFlag();
UpdateAllViews(NULL);
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Running the AutoServer Stand-alone Application

You've now created the stand-alone AutoServer application. To compile the program, choose Developer Studio's **Build, Build** command. Then, choose **Build, Execute** to run the program. When you do, you see the window shown in Figure 28.4. To see the application in action, choose the **Set, Set X Position** menu command. The display string moves to a new location (see Figure 28.5). When you choose the command again, the string moves back to its original position.



**FIG. 28.4** This is the AutoServer application's main window.



**FIG. 28.5** Here's the main window after the user chooses the **Set X Position** command.

AutoServer isn't exactly an award-winning application. (Talk about stating the obvious!) It is, however, fine for demonstrating how OLE automation works. Still, although AppWizard built a lot of OLE automation functionality into the program when you created it, you need to add the automation code that is specific to the application. In the following sections, you'll examine the code created by AppWizard. You'll then finish the automation server application by adding the properties and methods that make up the application's automation interface.

## Exploring the *CAutoServerApp* Class

To understand the source code, the first place to look is in `InitInstance()`, where the application performs its initialization at startup. That function, which is defined as part of the `CAutoServerApp` class, performs its normal initialization, as well as some special initialization for OLE. To initialize the OLE libraries, `InitInstance()` calls `AfxOleInit()`, as shown in Listing 28.2.

### Listing 28.2 `lst28_02.cpp`—Initializing ActiveX

```

if (!AfxOleInit())
{
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}
  
```

The `InitInstance()` function also calls the `AfxEnableControlContainer()` global function, which enables the application to act as a container for ActiveX controls. That function call looks like this:

```
AfxEnableControlContainer();
```

As you now know, all ActiveX and OLE applications must be registered with the system. Each application

and document type is identified by a GUID. In the case of the AutoServer application, the GUID is defined, as shown in Listing 28.3.

---

**Listing 28.3 lst28\_03.cpp—The Application’s GUID**

---

```
// {46DCFE58-6FBB-11D0-847F-444553540000}
static const CLSID clsid =
    { 0x46dcfe58, 0x6fbb, 0x11d0,
      { 0x84, 0x7f, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };
```

---

---

**NOTE:**

If you’ve followed the previous steps to create your own copy of the AutoServer application, your version will have a different GUID than the one shown in Listing 28.3.

---

Normally, when an automation server is loaded by a client application, the server’s window remains hidden. The `InitInstance()` function handles this detail by checking whether or not the application is being run as an OLE server. If so, the function registers all servers as running and returns without showing the main window. Listing 28.4 shows the program lines that perform these actions.

---

**Listing 28.4 lst28\_04.cpp—Initializing the Application as an OLE Server**

---

```
// Check to see if launched as OLE server
if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
{
    // Register all OLE server (factories) as running. This enables the
    // OLE libraries to create objects from other applications.
    COleTemplateServer::RegisterAll();

    // Application was run with /Embedding or /Automation. Don't show the
    // main window in this case.
    return TRUE;
}
```

---

When the application is run as a stand-alone application, the program skips over the lines in Listing 28.4 and instead registers the application in the Registry and displays the main window, as shown in Listing 28.5.

---

**Listing 28.5 lst28\_05.cpp—Registering the Application and Displaying Its Window**

---

```
// When a server application is launched stand-alone, it is a good idea
// to update the system registry in case it has been damaged.
m_server.UpdateRegistry(OAT_DISPATCH_OBJECT);
COleObjectFactory::UpdateRegistryAll();

// The one and only window has been initialized, so show and update it.
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
```

---

---

**NOTE:**

If you want the application’s window to appear even when it’s run as an OLE server, you can rewrite `InitInstance()` so that it displays the window no matter how the application is run. Another way to show the window at startup is to provide a `ShowWindow()` method in the server’s automation interface, and then call `ShowWindow()` from the client application. You’ll see how to create a `ShowWindow()` method later in this chapter, in the section entitled “Adding Properties and Methods to the Automation Server.”

---

## Exploring the *CAutoServerDoc* Class

In an MFC automation server application, it's the document class that handles the automation interface. First, the document class defines its own GUID, as shown in Listing 28.6.

### Listing 28.6 *lst28\_06.cpp*—Defining the Document Class's GUID

---

```
// {46DCFE82-6FBB-11D0-847F-444553540000}
static const IID IID_IAutoServer =
    { 0x46dcfe82, 0x6fbb, 0x11d0,
      { 0x84, 0x7f, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };
```

---

#### NOTE:

If you've created your own copy of the AutoServer application, your document class will have a different GUID than the one shown in Listing 28.6.

---

The document class also defines a dispatch map, which works similarly to the message maps you've used before, except that the dispatch map links the automation server's interface with the properties and methods that a client application can manipulate. At this point in the building of the application, the document class's dispatch map is defined, as shown in Listing 28.7. When you add properties and methods to the interface, ClassWizard will add entries to the dispatch map.

### Listing 28.7 *lst28\_07.cpp*—The Document Class's Dispatch Map

---

```
BEGIN_DISPATCH_MAP(CAutoServerDoc, CDocument)
    //{AFX_DISPATCH_MAP(CAutoServerDoc)
    // NOTE - the ClassWizard will add and remove mapping macros
    //      DO NOT EDIT what you see in these blocks of generated
    //}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Although the application class (CAutoServerApp, in this case) is responsible for initializing OLE, it is the document class (CAutoServerDoc, in this case) that sets up OLE automation. It does this in its constructor, as shown in Listing 28.8. The call to EnableAutomation() initializes automation for the application, whereas the call to the global function AfxOleLockApp() ensures that the server will remain in memory as long as a client application requires the server's services.

#### Listing 28.8 lst28\_08.cpp—Initializing OLE Automation

```
CAutoServerDoc::CAutoServerDoc()
{
    // TODO: add one-time construction code here

    EnableAutomation();

    AfxOleLockApp();
}
```

As you might have guessed, the call to AfxOleLockApp() has a complementary function that's called from the document class's destructor, as shown in Listing 28.9. The call to AfxOleUnlockApp() releases the server from its connection with this client. If no other clients are using the server, the server can be removed from memory.

#### Listing 28.9 lst28\_09.cpp—The Document Class's Destructor

```
CAutoServerDoc::~CAutoServerDoc()
{
    AfxOleUnlockApp();
}
```

### Adding Properties and Methods to the Automation Server

Now that you know how the basic application works, it's time to make AutoServer into a complete automation server. To create the full automation server application, you must create an interface that other application's can use to control the server. This interface is comprised of properties (similar to variables) and methods (similar to functions). The automation client (the application that controls the automation server) accesses values through the properties and performs actions

by calling the server interface's methods. In this section, you'll add a property and method to the AutoServer application. Just complete the following steps.

The complete source code and executable file for this version of the ActiveXServ application is located in the Chap28\AutoServer, Part 2 folder on this book's Web CD-ROM.



1. Display ClassWizard (press Ctrl+W), and choose the Automation page (see Figure 28.6). Choose CAutoServerDoc in the Class Name box.



**FIG. 28.6** ClassWizard's Automation page enables you to add properties and methods to the automation server's interface.

2. Click the Add Property button. The Add Property dialog box appears.
3. Make sure that you choose the Get/Set Methods option. Fill in the dialog box as shown in Figure 28.7. Click OK to dismiss the Add Property dialog box.

This new property will represent the document class's `m_x` variable, enabling an automation client application to change the position of the display string in the server's window. The `x` variable is what the interface's user sees as the property's name. The `m_x` data member is where the value of the `x` property is stored.

4. Click the Add Method button. The Add Method dialog box appears.
5. Fill in the dialog box as shown in Figure 28.8. Click OK to dismiss the Add Method dialog box.

The `ShowWindow()` method enables an automation client application to display the server application's window. Normally, a server application runs without displaying its window.



**FIG. 28.7** Create the automation interface's new property as shown here.



**FIG. 28.8** Create the automation server's new method as shown here.

6. Click the Edit Code button to find the `ShowWindow()` method. Add the lines shown in Listing 28.10 to the `ShowWindow()` method.

**Listing 28.10** `lst28_10.cpp`—Program Lines for the `ShowWindow()` Method

---



```

// Get a pointer to the view window.
POSITION position = GetFirstViewPosition();
CView* pView = GetNextView(position);

// If the window exists...
if (pView != NULL)
{
    // Get a pointer to the frame window.
    CFrameWnd* pFrameWnd = pView->GetParentFrame();

    // Show the frame window.
    pFrameWnd->ActivateFrame(SW_SHOW);
}

```

---

This code first gets a pointer to the server's view window. It then uses the view pointer to obtain a pointer to the server's parent window. Finally, the `ActivateFrame()` method is called through the parent window's pointer, which causes the window to be displayed.

**7.** Replace the return line in the `GetX()` property function with the following line:

```
return (short)m_x;
```

Returning the value as a short integer ensures that the server will work under both 16-bit and 32-bit Windows.

**8.** Complete the `SetX()` property function with the following program lines:

```

m_x = nNewValue;
SetModifiedFlag();
UpdateAllViews(NULL);

```

You've now added the specific automation functionality required by the `AutoServer` application. To compile the program, choose Developer Studio's Build, Build command. Now that you've completed this portion of the program, you've got an automation server that can also function as a stand-alone application. Unfortunately, you currently have no way to test the program's automation capabilities, because you have no automation client application that can use `AutoServer`'s automation interface. In the next section, you'll create that client application.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Creating an Automation Client Application

There are a number of ways that you can test an automation server, because the server can be accessed from any language that supports OLE automation. This includes not only C++, but also Visual Basic and Visual Basic for Applications. Because this is a C++ book, you'll now discover how to create a C++ client program using AppWizard. Creating an automation client application consists of a number of basic steps:

1. Create a normal AppWizard skeleton program.  
You don't have to select any OLE options when creating the skeleton application. Because the application requires so little OLE code, it's easier to add the lines you need than to remove a lot of lines you don't need.
2. Call `AfxOleInit()` from the application class's `InitInstance()` function.  
All application's that use ActiveX must call `AfxOleInit()` to initialize the OLE system.
3. Use ClassWizard to create an interface class from the server application's type library.  
Although you didn't know it, when you created the server application, ClassWizard created a type-library file for the server's interface. This type library includes the names and types of properties, as well as a list of the methods that a client application can call. ClassWizard takes the information in the type library and creates a class for the interface. This class makes it easy for you to access server properties or to call server methods.
4. Create, as a data member of the view class, an object of the interface class.  
Just as with any class, before you can access the class's data and function members, you have to create an instance of the class.
5. Call the interface object's `CreateDispatch()` function to load the automation server.  
Creating an instance of the interface class isn't enough to get it rolling. You also need to call `CreateDispatch()`, which loads the server and connects the interface object with the server's interface.
6. Access the server's interface through the interface object.  
After you have the interface object created and connected to the server's interface, you can access the server's properties and methods almost exactly as if they were part of your own application.

The requirements listed previously are easier to implement than they might sound, as you'll see as you create the automation client application. Complete the following steps to create the AutoClient application:



The complete source code and executable file for the AutoClient application is located on this book's CD-ROM.

1. Start a new AppWizard project workspace called AutoClient.

2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 28.9.

Step 1: Single document

Step 2: Default settings

Step 3: Default settings

Step 4: Default settings

Step 5: Default settings

Step 6: Default settings



**FIG. 28.9** The final project settings should look like this.

3. Bring up ClassWizard, and click the Add Class button and choose the From a Type Library selection (see Figure 28.10). The Import From Type Library dialog box appears.



**FIG. 28.10** You can use ClassWizard to create a class from a type library.

4. Use the dialog box to locate and select the AutoServer.tlb file in your project's Debug or Release directory (see Figure 28.11).



**FIG. 28.11** The .tlb file contains all of the information that ClassWizard needs about the automation server's interface.

5. Click the Open button, and the Confirm Classes dialog box appears (see Figure 28.12).

6. Click OK to accept the default names for the server interface class and its files. ClassWizard creates the new IAutoServer interface class. Click OK to close ClassWizard.

IAutoServer is the interface class that you'll use to connect to the server's interface.

7. Open the AutoClientView.h header file, and add the following line to the Attributes section, right after the public keyword:

```
IAutoServer m_autoServer;
```

This line creates an object of the IAutoServer class as a data member of the view class.



**FIG. 28.12** The Confirm Classes dialog box gives you a chance to change class and file names.

8. Type the following line near the top of the `AutoClientView.h` file, right before the class's declaration begins:

```
#include "AutoServer.h"
```

This line ensures that the `IAutoServer` interface class's declaration is accessible to the view class.

9. Use ClassWizard to add an `OnCreate()` function to the `CAutoClientView` class, as shown in Figure 28.13.



**FIG. 28.13** Add the `OnCreate()` function using these ClassWizard settings.

10. Click the Edit Code button, and add the lines shown in Listing 28.11 to the `OnCreate()` function, right after the `TODO: Add your specialized creation code here` comment:

---

**Listing 28.11** `lst28_11.cpp`—Program Lines for the `OnCreate()` Function

---

```
// Load the automation server object.
BOOL success =
    m_autoServer.CreateDispatch("AutoServer.Document");

// If AutoServer object didn't load, halt the program.
if (!success)
{
    AfxMessageBox("Failed to create AutoServer object!");
    return -1;
}
```

---

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

**11.** Use Developer Studio's menu editor to add the Automate menu shown in Figure 28.14. Give the Set X Position command the ID\_AUTOMATE\_SETXPOSITION menu ID, and give the Show Window command the ID\_AUTOMATE\_SHOWWINDOW menu ID.



**FIG. 28.14** The new Automate menu should look like this.

**12.** Use ClassWizard to create the OnAutomateSetxposition() message response function (see Figure 28.15).



**FIG. 28.15** Create the OnAutomateSetxposition() function like this.

**13.** Click the Edit Code button, and add the lines shown in Listing 28.12 to the OnAutomateSetxposition() function, right after the TODO: Add your command handler code here comment:

**Listing 28.12 lst28\_12.cpp—Program Lines for the OnAutomateSetxposition() Function**

```
// Get the current X position.
int x = m_autoServer.GetX();

// Set the new X position.
if (x == 20)
    m_autoServer.SetX(300);
else
    m_autoServer.SetX(20);
```

**14.** Use ClassWizard to create the OnAutomateShowwindow() message response function (see Figure 28.16).



**FIG. 28.16** Create the OnAutomateShowwindow() function like this.

**15.** Click the Edit Code button, and add the line shown below to the OnAutomateShowwindow() function, right after the TODO: Add your command handler code here comment:

```
m_autoServer.ShowWindow( );
```

**16.** Load the AutoClient.cpp file, and add the lines shown in Listing 28.13 to the beginning of the InitInstance() function:

**Listing 28.13 lst28\_13.cpp—Program Lines for the InitInstance() Function**

---

```
if (!AfxOleInit())
{
    AfxMessageBox("OLE initialization failed!");
    return FALSE;
}
```

---

**17.** Load the MainFrame.cpp file, and add the lines that follow to the PreCreateWindow() function:

```
cs.cx = 250;
cs.cy = 200;
```

These lines set the initial size of the application's frame window.

## Running the AutoClient Application

You've now created the AutoClient application. To compile the program, choose Developer Studio's Build, Build command. Then, choose Build, Execute to run the program. When you do, you see the window shown in Figure 28.17. To see the server application's window, choose the Automate, Show Window command. The server application's window appears, as shown in Figure 28.18.



**FIG. 28.17** Here's AutoClient when you first run it.





**FIG. 28.18** The client application can display the server application's window.

To change the server application's string position, choose the Automate, Set X Position command. This forces the server application to update its `m_x` member variable and redraw the string at its new position (see Figure 28.19).



**FIG. 28.19** The client application can also reposition the string displayed in the server application's window.

The automation server and client applications presented in this chapter only scratch the surface of what this powerful technology can do. You should use the knowledge you've gained here and spend some time creating automation applications that actually do useful work.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

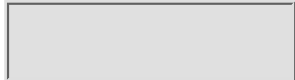
Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief
 Full

+ [Advanced Search](#)

+ [Search Tips](#)



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Chapter 29

# Creating ActiveX Controls

- Learn what an ActiveX control is and how it works
- Discover how to use ActiveX ControlWizard to create a skeleton control
- Learn how to create a control’s user interface
- See how to create properties for a control
- Learn how to add custom methods to a control
- Discover how to add ActiveX controls to Web pages

The hottest thing these days on the Internet is ActiveX controls, which not only can be included as objects in a container application’s document, but also can be embedded into Web pages to provide executable content on the Internet. Although creating an ActiveX control can be a complicated task, the basics are easy to master, especially considering Visual C++ 5.0 includes the MFC ActiveX ControlWizard, which performs much of the work for you. Then, you need only to add the specific program lines for your control. This chapter introduces you to creating ActiveX controls with ActiveX ControlWizard.

## Introducing ActiveX Controls

As far as the Internet is concerned, on the surface ActiveX controls are a lot like Java applets in that ActiveX controls enable Web developers to include mini-applications in their Web pages. However, although the results might look similar, on the inside ActiveX controls are nothing at all like Java applets.

What do ActiveX controls look like? Virtually, anything at all. Figure 29.1, for example, shows an application using Address ActiveX controls. The application actually contains two controls, each an instance of the Address control. One instance of the control is being used to retrieve an old address from the user and the other is used to get the new address.



**FIG. 29.1** The Address control provides a form in which a user can type in address information.

The Address control is a fairly simple example. ActiveX controls can be as sophisticated as you want them to be. A case in point is Microsoft's Calendar control (see Figure 29.2), which demonstrates various aspects of using and creating controls. The control's display includes 3D graphics, as well as a number of options that can be set by the user.



**FIG. 29.2** The Calendar control is a more sophisticated example of what ActiveX controls can do.

When you come right down to it, although most ActiveX controls perform simple tasks, an ActiveX control can also be a full-fledged application. From a user's point of view, the only difference between an ActiveX control and some other type of application is that ActiveX controls cannot survive on their own. That is, whereas you can run an application directly, an ActiveX control must be included as part of another application, Web page, or other type of ActiveX container. This is a pretty minor limitation, though, because an application running in its own window and an application containing an ActiveX control can look and act virtually identical to one another.

## Creating ActiveX Controls with MFC

If you've ever tried to program ActiveX controls before, you know that, unless you're an OLE expert, the programming involved can be a nightmare. OLE, on which ActiveX is based, is described in programming references thousands of pages long. Just trying to plow through this documentation to make sense of it is a full-time job even for professional programmers. Microsoft knows this, and so needed a way to bring the programming of ActiveX controls down to a level that the average person can understand. Without making ActiveX more accessible, there was no way Microsoft would be able to compete with Sun Microsystems's Java.

Enter MFC and ActiveX ControlWizard, which hides almost all of the difficult details of creating ActiveX controls, enabling the programmer to spend his time making controls, rather than trying to figure out indecipherable programming manuals. Creating ActiveX controls with MFC and ControlWizard can actually be easier than creating applets with Java. Anyone with basic programming knowledge can get started with ControlWizard immediately.

## Creating the Basic Calculator ActiveX Control

The first step in creating a control is to run MFC ControlWizard to create the skeleton program code for the control. This program code provides all of the basics needed to compile and run an ActiveX control. Before the control will

do what you want, however, you have to add your own custom program lines, which you'll do throughout the remainder of this chapter. For now, complete the following steps to create the basic Calculator control.

The complete source code and executable file for this version of the ActiveXServ application is located in the Chap29\Calculator, Part 1 folder on this book's CD-ROM.



1. Start a new MFC ActiveX ControlWizard project workspace called Calculator.
2. Give the new project the following settings in the ControlWizard dialog boxes. The New Project Information dialog box should then look like Figure 29.3.

Step 1: Default Settings

Step 2: Default settings



**FIG. 29.3** These are the project settings for the Address control.

## Running the Basic Calculator Control

Believe it or not, you've now created the basic Calculator application. To compile the control, choose Developer Studio's Build, Build command. After compiling the control, Visual C++ also registers it with the system. To prove that Calculator is already a runnable control, you can load the control into Microsoft's test container application. To do this, choose Developer Studio's Tools, ActiveX Control Test Container command. When you do, the test container application appears.

In the test application, choose the Edit, Insert OLE Control command. The Insert Control dialog box appears (see Figure 29.4). Double-click Calculator Control to embed the new Calculator control into the test application's document. The control appears in the application's window, as shown in Figure 29.5.



**FIG. 29.4** You use the Insert Control dialog box to choose the control to embed.



**FIG. 29.5** Here's the basic Calculator control embedded in the test application's document.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 ● [Advanced](#)  
 ● [Search](#)  
 ● [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

As you can see, right now the Calculator control doesn't look like much. In fact, its appearance is created totally by the default source code provided by ControlWizard. To change the control's appearance and functionality, you must modify the skeleton control that was created by the wizard. In the following section, you take care of that task. About the only thing you can do with the default control is move it around the window, resize it, and display its About dialog box.

To display the About dialog box, first make sure that the control is selected (click it), and then choose the test application's Edit, Invoke Methods command. The Invoke Control Method dialog box appears, as shown in Figure 29.6. In the Name text box, you can select the method that you want to invoke. Currently, the control has only a single method called AboutBox. Just click the Invoke button to invoke the method. The control's About dialog box appears. Close the dialog box, and then click the Close button to dismiss the Invoke Control Method dialog box. Now, you can close the test application.



**FIG. 29.6** The Invoke Control Method dialog box enables you to test your control's methods.

## Creating the Control's User Interface

There are a few special things that you need to know to modify the basic ActiveX control created by ControlWizard. However, much of what you already know about MFC can be applied toward the control. For example, just like most MFC programs, in order to draw the control's appearance, you need to write program lines for the OnDraw() function. In this section, you'll create a user interface for the Calculator control by completing the following steps.

The complete source code and executable file for this version of the Calculator control is located in the Chap29\Calculator, Part 2 folder on this book's CD-ROM.



1. Load the CalculatorCtl.h header file, and add the lines shown in Listing 29.1 to the class's declaration, right after the line CCalculatorCtrl();.

### Listing 29.1 lst29\_01.cpp—New Data Members for the CCalculatorCtrl Class

```

CEdit m_value1;
CEdit m_value2;
CEdit m_result;
CButton m_calculate;
  
```

These lines declare the four controls that make up the Calculator control's user interface.

2. Use ClassWizard to add the OnCreate() function to the CCalculatorCtrl class, as shown in Figure 29.7.



**FIG. 29.7** The OnCreate() function responds to the WM\_CREATE Windows message.

3. Click the Edit Code button, and add the lines shown in Listing 29.2 to the OnCreate() function, right after the TODO: Add your specialized creation code here comment:

---

**Listing 29.2 lst29\_02.cpp—Program Lines for the OnCreate() Function**

---

```
m_value1.Create(WS_CHILD | WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL,
    CRect(10, 60, 110, 90), this, IDC_VALUE1);
m_value2.Create(WS_CHILD | WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL,
    CRect(10, 100, 110, 130), this, IDC_VALUE2);
m_result.Create(WS_CHILD | WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL,
    CRect(120, 100, 220, 130), this, IDC_RESULT);
m_calculate.Create("Calculate", WS_CHILD | WS_VISIBLE |
    WS_BORDER | BS_PUSHBUTTON,
    CRect(120, 60, 220, 90), this, IDC_CALCULATE);
```

---

These lines create the controls that make up the Calculator control's user interface.

4. Choose Developer Studio's View, Resource Symbols command. The Resource Symbols dialog box appears, displaying the resource IDs currently defined in your project.
5. Click the New button. The New Symbol dialog box appears. In the Name box, type **IDC\_VALUE1** (see Figure 29.8), and click OK.  
You just added the IDC\_VALUE1 resource ID to the project. In Listing 29.2, you can see that this ID is assigned to the first edit control, called m\_value1.



**FIG. 29.8** You need to create resource IDs for Calculator's edit and button controls.

6. Repeat Step 5 to create IDs for the remaining controls. The IDs should be IDC\_VALUE2, IDC\_RESULT, and IDC\_CALCULATE and should have the values 102, 103, and 104, respectively. Click the Close button to dismiss the Resource Symbols dialog box.
7. In the OnDraw() function, delete the line pdc->Ellipse(rcBounds);.  
This is the line that draws the control's default display, which is a shaded ellipse. The Calculator control will create its own custom display.
8. Add the lines shown in Listing 29.3 to the end of the OnDraw() function, right after the line pdc->FillRect(rcBounds, CBrush::FromHandle((HBRUSH)GetStockObject(WHITE\_BRUSH)));:

---

**Listing 29.3 lst29\_03.cpp—New Program Lines for the OnDraw() Function**

---

```
// Print labels for the controls.
pdc->TextOut(10, 40, "Enter Values Below");
pdc->TextOut(150, 130, "Result");

// Display the Calculator control's caption in a large font.
CFont font;
font.CreateFont(30, 0, 0, 0, FW_BOLD, 0, 0, 0, DEFAULT_CHARSET,
```

```
OUT_CHARACTER_PRECIS, CLIP_CHARACTER_PRECIS, DEFAULT_QUALITY,  
DEFAULT_PITCH | FF_DONTCARE, "TimesRoman");  
CFont* oldFont = pdc->SelectObject(&ampfont);  
pdc->TextOut(10, 0, "Calculator");  
pdc->SelectObject(oldFont);
```

---

You've now created the Calculator control's user interface. To compile the control, choose Developer Studio's **B**uild, **B**uild command. After compiling the control, Visual C++ also registers it with the system. To see the Calculator control's new user interface, load the control into Microsoft's test container application. After embedding the control into the test application's document, you'll need to resize the control, as shown in Figure 29.9.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Tip:

To register a control without rebuilding it, choose the Tools, Register Control command in Developer Studio's menu bar. This command enables you to register a control located at this book's Web site.



**FIG. 29.9** The Calculator control's display includes edit and button controls.

## Adding Properties to the Control

As do other types of ActiveX objects, ActiveX controls can feature properties and methods. As you already know, properties are similar to data members, and methods are similar to functions. In most cases, you enable the user to manipulate the control's properties and methods, so that he can customize how the control looks and acts in his application or Web page. In this section, you'll add two types of properties to the Calculate control. The first type of property is a data member that can be handled using a notification method. The second type of property is a value that is manipulated by Get() and Set() property methods. Complete the following steps to add properties to the Calculator control.



The complete source code and executable file for this version of the Calculator control is located in the Chap29\Calculator, Part 3 folder on this book's CD-ROM.

1. Press Ctrl+W to display ClassWizard, and select the wizard's Automation page.
  2. Click the Add Property button. The Add Property dialog box appears. Fill in the dialog box as shown in Figure 29.10. Click the OK button to add the property to the control.
- You just created a property called MultValues, which is a Boolean value. The value of the property is stored in a data member called m\_multValues. The MultValues property will control whether the Calculator control sums or multiplies the values entered by the user.



**FIG. 29.10** This is how you create a property that's associated with a notification method.

3. Click the Add Property button again. This time fill in the Add Property dialog box as shown in Figure 29.11. Make sure the Get/Set Methods option is checked, and click the OK button to add the property to the controls.

You just created a property called ControlCaption, which is a string value. (Actually, it's a BSTR value, which is a special string type used with automation data manipulation functions.) The value of the property is controlled by Get() and Set() property methods, enabling the control to check changes to the property. The ControlCaption property will control the main caption displayed in the Calculator control.



**FIG. 29.11** This is how you create a property that is accessed through Get() and Set() property methods.

4. Click the Add Property button. Fill in the Add Property dialog box as shown in Figure 29.12. Make sure the Get/Set Methods option is checked, and click the OK button to add the property to the controls. Finally, close ClassWizard by clicking the OK button.

You just created a property called ButtonCaption, which is a string value. The value of the property is controlled by Get() and Set() property methods, enabling the control to check changes to the property. The ButtonCaption property controls the caption displayed by the Calculator control's button.



**FIG. 29.12** The ButtonCaption property determines the button's label.

5. Load the CalculatorCtl.h header file, and add the following lines to the class's declaration, right after the line CButton m\_calculate, which you placed there previously:

```
CString m_buttonCaption;  
CString m_controlCaption;
```

These lines declare string objects as data members of the CCalculatorCtrl class. These strings will hold the values of the ButtonCaption and ControlCaption properties. When you create properties that are controlled by Get() and Set() property methods, you must manually add storage for the properties to your

control's class.

6. Load the CalculatorCtl.cpp file, locate the OnDraw() function, and replace the line `pdcc->TextOut(10, 0, "Calculator")` with the following:

```
pdcc->TextOut(10, 0, m_controlCaption);
```

Now the control's caption will be the string stored in the `ControlCaption` property (which is stored in the `m_controlCaption` data member), rather than the static string "Calculator."

7. Locate the `OnCreate()` function, and replace the lines

```
m_calculate.Create("Calculate", WS_CHILD | WS_VISIBLE |  
    WS_BORDER | BS_PUSHBUTTON,  
    CRect(120, 60, 220, 90), this, IDC_CALCULATE);
```

with

```
m_calculate.Create(m_buttonCaption, WS_CHILD |  
    WS_VISIBLE | WS_BORDER | BS_PUSHBUTTON,  
    CRect(120, 60, 220, 90), this, IDC_CALCULATE);
```

Now the button's caption will be the string stored in the `ButtonCaption` property (which is stored in the `m_buttonCaption` data member), rather than the static string "Calculator."

8. Locate the `DoPropExchange()` function, and add the following lines right after the `TODO`: Call `PX_` functions for each persistent custom property comment:

```
PX_Bool(pPX, "MultValues", m_multValues, FALSE);  
PX_String(pPX, "ControlCaption", m_controlCaption,  
    "Calculator");  
PX_String(pPX, "ButtonCaption", m_buttonCaption,  
    "Calculate");
```

MFC ActiveX controls, much like MFC dialog boxes, feature a mechanism that automatically initializes and transfers the values of the control's properties. These values are even saved automatically so that they can be restored whenever the control is reloaded. Each such property, called a ***persistent property***, must have a `PX_` line in the `DoPropExchange()` function. You'll learn more about persistent properties later in this chapter, in the section entitled "Understanding Persistent Properties."

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



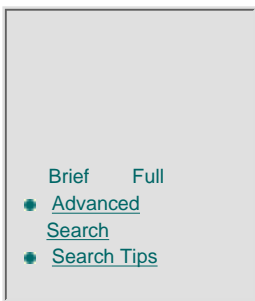
FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGEBROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

**9.** Add the following line to the GetControlCaption() method, right after the TODO: Add your property handler here comment:

```
strResult = m_controlCaption;
```

This line sets the returned string to the current value of the ControlCaption property, which is stored in the m\_controlCaption data member.

**10.** Add the following line to the GetButtonCaption() method, right after the TODO: Add your property handler here comment:

```
strResult = m_buttonCaption;
```

This line sets the returned string to the current value of the ControlCaption property, which is stored in the m\_controlCaption data member.

**11.** Add the following lines to the SetControlCaption() method, right after the TODO: Add your property handler here comment:

```
m_controlCaption = lpszNewValue;  
InvalidateControl();
```

These lines store the new value of the ControlCaption property and redraw the control so that it shows the current ControlCaption string.

**12.** Add the following lines to the SetButtonCaption() method, right after the TODO: Add your property handler here comment:

```
m_buttonCaption = lpszNewValue;  
m_calculate.SetWindowText(m_buttonCaption);
```

These lines store the new value of the ControlCaption property and redraw the button's text so that it shows the current ButtonCaption string.

## Testing the Calculator Control's Properties

You've now added properties to the Calculator control. To compile the control, choose Developer Studio's Build, Build command. After compiling the control, Visual C++ also registers it with the system. To see the Calculator control's new properties in action, load the control into Microsoft's test container application. After embedding the control into the test application's document, you'll need to resize the control.

Currently, the control is displaying the default values for the ControlCaption and ButtonCaption properties. Further, the MultValues property is set to its default value of FALSE. To see that this is true, first make sure the control is selected (click it, if it's not), and then choose the test application's View, Properties command. The Properties dialog box appears. Choose a property in the Property box, and its current setting appears in the Value box (see Figure 29.13).



**FIG. 29.13** The Property dialog box enables you to view and edit property settings.

To change a property, choose it in the Property box, type its new setting in the Value box, and then click the Apply button. When you do, the control changes to show the new property setting. Figure 29.14 shows what happens if you change the `ControlCaption` property to “Adding Machine.” You can change the `ButtonCaption` property in exactly the same way.



**FIG. 29.14** When you change a property, the control immediately uses the new property value.

## Understanding Persistent Properties

As you learned previously, persistent properties automatically retain their values. For this to work, you must write, for each property, a `PX_` program line in the `DoPropExchange()` function. In the Calculator control, the `DoPropExchange()` function looks like Listing 29.4. As you can see in the function, each property has a `PX_` program line. Each of these `PX_` lines reflects the data type for the property. For example, the Boolean `MultValues` property is represented by the `PX_Bool()` line.

### Listing 29.4 `lst29_04.cpp`—The Calculator Control’s `DoPropExchange()` Function

---

```
void CCalculatorCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);

    // TODO: Call PX_ functions for each persistent custom property.

    PX_Bool(pPX, "MultValues", m_multValues, FALSE);
    PX_String(pPX, "ControlCaption", m_controlCaption, "Calculator");
    PX_String(pPX, "ButtonCaption", m_buttonCaption, "Calculate");
}
```

---

There are 16 `PX_` functions that you can use:

- `PX_Blob()`
- `PX_Bool()`
- `PX_Color()`
- `PX_Currency()`
- `PX_DataPath()`
- `PX_Double()`
- `PX_Float()`
- `PX_Font()`
- `PX_IUnknown()`
- `PX_Long()`
- `PX_Picture()`

- `PX_Short()`
- `PX_String()`
- `PX_ULong()`
- `PX_UShort()`
- `PX_VBFontConvert()`

These functions have four arguments, which are a pointer to the `CPropExchange` object (passed into `DoPropExchange()` as a parameter), the property name, the property's storage variable, and the property's default value. If the property were previously set, its value is restored. Otherwise, the property gets the default value.

## Property Notification Functions

Two of the Calculator control's properties, `ControlCaption` and `ButtonCaption` are managed by `Get()` and `Set()` methods. The `MultValues` property, however, has no `Get()` or `Set()` methods. Instead, there is a notification method, called `OnMultValuesChanged()`, which is called whenever the `MultValues` property changes. The default `OnMultValuesChanged()` looks like Listing 29.5. The default function calls `SetModifiedFlag()` so that the control knows that it needs to save its persistent properties. You can use the notification function for other purposes, as well, such as validating the new property value.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

### Listing 29.5 lst29\_05.cpp—The Default Implementation of OnMultValuesChanged()

```
void CCalculatorCtrl::OnMultValuesChanged( )
{
    // TODO: Add notification handler code

    SetModifiedFlag( );
}
```

## Adding a Button Notification Function

At this point in its programming, the Calculator control's visual user interface is complete. However, the control still doesn't do anything useful. When the user enters two values and clicks the control's button, the control should perform the calculation and display the result. In this section, you add the program lines that accomplish this task by completing the following steps.



The complete source code and executable file for this version of the Calculator control is located in the Chap29\Calculator, Part 4 folder on this book's CD-ROM.

1. Load the CalculatorCtl.h header file, and add the following line to class's message map declaration, right before the DECLARE\_MESSAGE\_MAP() line:

```
afx_msg void OnButtonClicked( );
```

This line declares the OnButtonClicked() function, which will respond to button clicks.

2. Load the CalculatorCtl.cpp file, and add the following line to the class's message map (found near the top of the file), right after the ON\_OLEVERB(AFX\_IDS\_VERB\_PROPERTIES, OnProperties) line:

```
ON_BN_CLICKED( IDC_CALCULATE, OnButtonClicked )
```

This line associates the OnButtonClicked() function with button-click notifications. That is, when the user clicks the control's Calculate button, MFC will call OnButtonClicked(), where the program can respond



to the button click.

3. Add the function shown in Listing 29.6 to the end of the file. This function, which responds to the Calculate button, retrieves the values the user typed into the text boxes, converts those values to integers, performs the required calculation (as determined by the value of the MultValues property), and displays the result.

---

**Listing 29.6 lst29\_06.cpp—The OnButtonClicked() Notification Function**

---

```
void CCalculatorCtrl::OnButtonClicked()
{
    CString s1, s2;
    m_value1.GetWindowText(s1);
    m_value2.GetWindowText(s2);
    int value1 = atoi(s1);
    int value2 = atoi(s2);

    int result;
    if (!m_multValues)
        result = value1 + value2;
    else
        result = value1 * value2;

    char s[20];
    wsprintf(s, "%d", result);
    m_result.SetWindowText(s);
}
```

---

## Testing the Button

You've now added the program lines needed for the Calculator control to perform its intended function. To compile the control, choose Developer Studio's **B**uild, **B**uild command. After compiling the control, Visual C++ also registers it with the system. To see the Calculator control in action, load the control into Microsoft's test container application. After embedding the control into the test application's document, resize the control so that it's fully displayed.

Now, type two integers into the control's text boxes, and click the Calculate button. The control sums the two values and displays the result (see Figure 29.15). Next, choose the test application's **V**iew, **P**roperties command. When the Properties dialog box appears, change the MultValues property to 1, as shown in Figure 29.16. Then, click the Calculate button again. This time the control finds the product of the two values, rather than the sum.



**FIG. 29.15** In its default state, the Calculator control sums the entered values.



**FIG. 29.16** By changing the MultValues property to TRUE (a nonzero value), the control will multiply values, rather than add them.

## Adding Methods to a Control

The Calculator control already has one method that can be accessed by a developer using the control in an application or a Web page. That method is the one that displays the control's About dialog box. However, the AboutBox() method was created by ControlWizard. Now, you'll learn to add your own methods to the Calculator control by completing the following steps to accomplish this important task.



The complete source code and executable file for this version of the Calculator control is located in the Chap29\Calculator, Part 5 folder on this book's CD-ROM.

1. Press Ctrl+W to display ClassWizard, and then select the wizard's Automation page.
2. Click the Add Method button. The Add Method dialog box appears.
3. Fill in the Add Method dialog box, as shown in Figure 29.17. Click the OK button to add the new ValidateValues() method to the control.



**FIG. 29.17** Here's how you add a method to a control.

4. Click the Edit Code button, and replace the lines in the ValidateValues() function with the lines shown in Listing 29.7:

### Listing 29.7 lst29\_07.cpp—Program Lines for the ValidateValues() Method

```
// Initialize the method's return code.
BOOL valuesOK = TRUE;

// Retrieve strings from the text boxes.
CString s1, s2;
m_value1.GetWindowText(s1);
```

```
m_value2.GetWindowText(s2);

// Check that at least the first character of
// the first string is a digit.
TCHAR c = s1.GetAt(0);
if (c < '0' || c > '9')
    valuesOK = FALSE;

// Check that at least the first character of
// the second string is a digit.
c = s2.GetAt(0);
if (c < '0' || c > '9')
    valuesOK = FALSE;

// If the values are no good, tell the user.
if (!valuesOK)
    MessageBox("Invalid values");

return valuesOK;
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Testing the Method

You've now added a new method to the Calculator control. To compile the control, choose Developer Studio's **Build, Build** command. After compiling the control, Visual C++ also registers it with the system. To test the Calculator control's new method, load the control into Microsoft's test container application. After embedding the control into the test application's document, resize the control so that it's fully displayed.

Now, type two integers into the control's text boxes, and choose the test application's **Edit, Invoke Methods** menu command. When the **Invoke Control Method** dialog box appears, choose **ValidateValues** in the **Name** box, and click the **Invoke** button. The test application calls the **ValidateValues()** method and displays the return value at the bottom of the **Invoke Control Method** dialog box. Because you typed valid digits into the text boxes, the return value is **TRUE** (see Figure 29.18).



**FIG. 29.18** If you type valid integers, the **ValidateValues()** method returns **TRUE**.

Now, close the **Invoke Control Method** dialog box, and change one of the entries in the control's text boxes to an invalid value, such as the word "one." Use the **Invoke Control Method** dialog box to invoke the **ValidateValues()** method again, and you'll see a message box informing you that you've entered invalid values (see Figure 29.19). Moreover, the **Invoke Control Method** dialog box displays a return value of **FALSE**.



**FIG. 29.19** The control can now warn users when they've entered invalid values.

## Adding an ActiveX Control to a Web Page

As you know, ActiveX controls can be used both as embedded objects in an application's document (such as when you added the control to the test application's window) and a control in a Web page. The first step in building a

Web page that contains ActiveX controls is including the controls in the HTML document. Microsoft created the new <OBJECT> tag for just this task.

The <OBJECT> tag is a lot like other HTML tags you might have run into before such as <IMG> and <APPLET>. The <OBJECT> tag is a little more sophisticated, however, and harder to decipher, because it must include the ActiveX control's **GUID**, which is a special ID for the control that's guaranteed to be unique. An operating system, such as Windows 95, uses the GUID in its registry to positively identify the control. An <OBJECT> tag for the Calculator control might look like Listing 29.8.

#### **Listing 29.8** lst29\_08.cpp—The <OBJECT> tag

---

```
<OBJECT
    classid="clsid:73E7511B-72B9-11D0-847F-444553540000"
    id=Calculator
    height = 150
    width = 235>
</OBJECT>
```

---

The <OBJECT> tag in Listing 29.8 tells the HTML document to find, load, and display the Calculator control. This form of the <OBJECT> tag contains a minimum amount of information needed to load and manipulate the control. The first part of the tag is the classid, which also happens to be comprised mainly of the control's GUID:

```
classid="clsid:73E7511B-72B9-11D0-847F-444553540000"
```

---

**Note:**

Because CLSIDs are unique from one control project to another, yours will be different than the one shown here.

---

How can you get the control's ID? This is pretty easy to do because Visual C++ has already registered the control in your system Registry. To find the ID, first run the REGEDIT.EXE tool, found in your main Windows directory. When you do, you see the window shown in Figure 29.20. Double-click the HKEY\_CLASSES\_ROOT folder, and then find the CALCULATOR.CalculatorCtrl.1 entry in the list that appears. Double-click this entry, and then click the CLSID entry. The control's GUID appears in the window's right pane (see Figure 29.21).

Returning to the <OBJECT> tag, the second part of the tag specifies the control's ID, which is a name that can be used within the HTML document to identify and communicate with the control. You would, for example, use the ID to manipulate the control from a script language like VBScript. The ID looks like this:

```
id=Calculator
```

Finally, the height and width parameters enable you to determine the size of the

control when it's loaded:

```
height = 150  
width = 235>
```



**FIG. 29.20** The REGEDIT tool enables you to view and edit the Registry.



**FIG. 29.21** The CLSID entry displays the control's GUID.

Listing 29.9 is an HTML document that loads and displays the Calculator control. If you load this HTML document into Microsoft Internet Explorer (currently, only Internet Explorer supports ActiveX controls), you'll see the window shown in Figure 29.22. Go ahead and use the control to sum some numbers.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE

BY TOPIC

## Part V

# Active Template Library (ATL)

## Chapter 30

# Using the Active Template Library

- Understanding ATL
- Comparing ATL and MFC
- Building an ATL Control

Most Windows programmers will tell you that learning to program with Microsoft's Component Object Model (COM) is a difficult, error-prone process. Until fairly recently, interest in COM was restricted to people who needed to make use of OLE in some form or another; MFC programmers largely had no need to use COM because MFC has provided OLE support classes for several releases.

COM programming, however, has become increasingly important. The explosive growth of interest in the Internet led Microsoft to restyle OLE controls as ActiveX, and ActiveX depends entirely on the use of COM interfaces. Furthermore, many of the new user-interface and shell features in Exchange, Win95, and Windows NT 4.0 are only available through the use of COM interfaces.

## Introducing the ActiveX Template Library

With the release of Visual C++ 5.0, the ActiveX Template Library (ATL) has matured into a complete ActiveX development solution. The 1.0 and 1.1 releases of ATL introduced ATL as an alternative to MFC for developing ActiveX controls. These early versions of ATL, however, only supported ActiveX COM objects and ActiveX Automation servers. Although these functions were useful in their own right, they didn't satisfy developers' demands for an easier way to write ActiveX controls.

There's long been an imbalance between OLE control developers and users—whereas the users have had easy-to-use visual tools like Visual Basic and Delphi, the control developers have been stuck writing their controls from scratch with little automated assistance.

ATL version 2.1, the version that ships with VC++ 5.0, begins to remedy that injustice by supporting ActiveX controls and providing an Object Wizard that can be used to create various types of ActiveX components.



<http://www.microsoft.com/visualc/prodinfo/atlinst.htm> ATL 2.1 is included with the current version of Visual C++ 5.0. If you're using an older version of VC++, you can still get the latest version of ATL from Microsoft's Web site; if you're using Visual C++ 4.0, 4.1, or 4.2, however, you must also upgrade to Visual C++ 4.2b to use ATL 2.1.

## How ATL Is Implemented

ATL is provided as a set of extensions to the core Visual C++ environment. The first, and most visible, component is the new "ATL COM AppWizard" item that appears in the Projects tab of the New Project dialog box, as shown in Figure 30.1. As with all of the other AppWizards that you've used throughout this book, the ATL COM AppWizard asks questions specific to ATL projects. We'll go into more detail about the ATL COM Wizard's questions in the section "Creating a New ATL Project."



**FIG. 30.1** The first step in building an ATL project is to tell VC++ to use the ATL COM AppWizard.

---

### Note:

The ATL source code is in `DevStudio\VC\ATL`; it's well worth a look if you're curious about how it was implemented, because it neatly combines some advanced C++ with familiar Windows and COM concepts.

---

**Templates for COM Objects and Interfaces.** The *T* in ATL stands for "template." C++ templates offer a powerful mechanism for abstracting data into classes. Instead of writing many separate classes for similar types of operations, you can write a *template class* that can be instantiated to handle many data types. For example, suppose you are writing an e-mail application and want to build classes to handle all of the structures shown in Listing 30.1.

### Listing 30.1 Sample Structures for Template Expansion

---

```
typedef struct
{0
    int            messageClassification;
    LPTSTR         senderName;
    LPTSTR         subject;
    . . .
} incomingMessage;
typedef struct
{
    int            messageClassification;
    CFileArray     attachedFiles;
    LPTSTR         subject;
    Recipient      *toRecipients;
    . . .
}
```



```

} outgoingMessageS;

typedef struct
{
    LPTSTR        mailboxPath;
    LPTSTR        mailboxName;
    long          messageCount;
    . . .
} mailboxS;

```

---

Without templates, you'd have three separate array classes—even though their implementations would be very similar. You could work around the problem by using an array of void pointers or handles, but then you'd lose the benefits of type safety and argument checking. With templates, you can use the template class like this (this example uses MFC's CList class):

```

CList<incomingMessageS, incomingMessageS> incomingMessageArr;
CList<outgoingMessageS, outgoingMessageS> outgoingMessageArr;
CList<mailboxS, mailboxS> mailboxArr;

```

Here, you're creating three new instances of CList. The template's two arguments are the type of data to store in the array and the type of data that the new array class's member functions should use as arguments, where appropriate. You can then use the template-based classes in your code; the compiler takes care of expanding your template definitions into three new classes, each with a full set of functions to handle your class.

ATL's more than just templates, though; the *L* in ATL is for the library code found in `\ProgramFiles\DevStudio\VC\ATL\Include`. This code serves a purpose similar to MFC: It implements all the complex behavior that ATL supports. The templates wrap the libraries so that they're easy to reuse, but the libraries take care of all the sticky COM- and OLE-oriented details for you.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

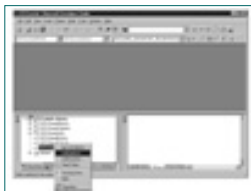
[Previous](#)
[Table of Contents](#)
[Next](#)

**ATL Object Wizard.** MFC provides the AppWizard and ClassWizard for building application functionality. When you want to add new methods or members to an MFC class, you use the ClassWizard. However, there's no ClassWizard support for the ATL; instead, you use the ATL Object Wizard dialog box, as shown in Figure 30.2, to create new ATL-based classes.

In MFC, you use the ClassWizard to add new members or functions. In ATL, you use the right-button context menu on objects in the ClassView tab of the Workspace window. As shown in Figure 30.3, the context menu enables you to add properties and methods to the control's class.



**FIG. 30.2** The ATL Object Wizard dialog box enables you to add new ATL-based objects to your project.



**FIG. 30.3** To add properties or methods to an ATL control, use the right mouse button.

**Automatic IDL Generation.** The Interface Definition Language, or IDL, provides a way for you to tell Windows what interfaces, properties, and classes your controls expose to the outside world, as well as what classes they depend on. IDL has a primarily C-like syntax, but writing IDL from scratch is a fearsomely complex task. One of Microsoft's key objectives in building ATL was to speed the ActiveX control development process by reducing the amount of IDL hacking required to produce ActiveX controls. Note that (as you'll see in the next chapter) you still have to manually edit the IDL file to support control events, but the ATL COM AppWizard provides a usable skeleton from the start.

- See "Adding the Event Interface" to learn more about manually editing the IDL file, **p. 657**

## What ATL Is

ATL is a lightweight library of templates designed to make it easy to build small, fast ActiveX controls. Of course, if you read Chapter 29, “Creating ActiveX Controls,” you might be asking why you’d even want to bother with ATL; the MFC ControlWizard does a good job of simplifying the process.

One good reason: Most users will accept software that includes several hundred K of MFC DLLs if it comes on floppy or CD, but baggage that large is excessive for controls designed to be downloaded over the Internet.

Performance is another area where ATL shines; the MFC message-passing architecture imposes overhead that ATL controls avoid. Because ATL is implemented as a set of templates, there’s very little runtime overhead for interface queries and passing.

The ATL Object Wizard hides most of the complexity of adding new properties and methods to your controls. In addition, the Object Wizard supports the full set of ambient and stock properties available to ActiveX controls.

- See “Using Properties” to learn more about ambient and stock properties, **p. 643**

## What ATL Isn’t

From the discussion thus far, you might think ATL is the biggest boon to Windows programming since `#define STRICT`. However, ATL’s not intended to be a general-purpose solution for writing any kind of programs; it has some limitations that you need to be aware of.

Despite what you might be led to think from its close association with MFC, ATL’s not an application framework. It doesn’t directly support menus, toolbars, document windows, printing, sockets, or most of the other MFC features that you’ve used in preceding chapters.

ATL is optimized for use with COM. Although you can accomplish a lot without a deep understanding of COM, real mastery of ATL will require you to spend some time getting your hands dirty with the COM interfaces implemented by ATL. This is of course no different from the learning process involved with MFC—you can get a lot done without knowing anything about the Win32 API, but at some point you have to move beyond the framework and call API routines directly.

---

### Tip:

If you want to learn more about COM, Kraig Brockschmidt’s *Inside OLE* is the standard reference work. It’s a thick, imposing book, but Brockschmidt clearly explains how COM works in great detail.

---

Furthermore, as you’ll see in Chapter 31, there’s still a good bit of manual labor involved in customizing ATL-based controls. Microsoft has largely succeeded in reducing the amount of IDL tweaking and hand-customization of COM interfaces needed to make ActiveX controls, but it hasn’t done away with it entirely. (Good news, though—you can take a working control like the

one used in Chapters 31 and 32 and reuse it over and over!)

## When to Use ATL

If you want to build COM interfaces and COM objects, ATL offers some significant advantages. With ATL, you can build fast, lightweight COM servers in a reasonable amount of time.

ATL is designed entirely to help you implement small, fast COM servers in C++. These servers can expose custom COM or IDispatch-based interfaces; however, ATL doesn't include support for more complex COM-based architectures, such as ActiveX documents.

ATL is best used for the following types of projects:

- ActiveX controls.
- Internet Explorer-only controls (like ActiveX controls, but without some of the general-purpose ActiveX interfaces).
- Generic COM or Distributed COM objects; you can use these to distribute processing over a network, or to encapsulate code that doesn't need a user interface.
- COM objects that need to support both IDispatch and custom COM interfaces.
- COM objects, where you want to use free threading (only available in Windows NT 4.0, as of this writing).

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief Full  
• [Advanced Search](#)  
• [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

If your project meets any of these criteria and doesn't have a significant user interface, then ATL is probably the best choice for producing the smallest, fastest code.

On the other hand, if you want to create complex servers that need to support user-interface items, embedded ActiveX controls, or full-blown ActiveX documents, you should use a more robust framework like MFC instead of ATL.

## ATL versus MFC

At first glance, ATL and MFC might seem very different. After all, MFC is a large and complex web of classes built to offer every imaginable application service, from printing to ODBC database access to providing a standard Windows user interface; in contrast, ATL is a small, highly focused set of services targeted at producing COM objects.

However, there's one critical area of agreement between the two: They were both designed to shift the burden of programming Windows from your shoulders onto their own. In this light, MFC and ATL are partners, not competitors.

You can see from the preceding sections that MFC and ATL both have their own niches. MFC's broad range of services and excellent integration with the Developer Studio make it easy to produce full-featured applications—but those applications can be bulky. ATL produces small, fast code, but it can't easily be made to do many things that MFC completely automates.

The good news, though, is that MFC and ATL can productively be used together. The first route is to use MFC classes in your ATL controls. Although this can simplify your programming job, it bogs down your controls with a lot of unneeded baggage. A better approach is to use MFC-based applications to host your controls. By using the techniques described in Chapter 26, "Creating an ActiveX Container Application," you can use MFC where it makes sense while still gaining the benefits of ATL for building controls and extension objects.

## Creating Your First ATL Project

The basic steps needed to create a new ATL project are similar to those needed to create any other kind of project in VC++. Because the ATL Object Wizard is a new addition, though, it's worth looking at its features to understand how, and when, to use them.

## Using the ATL COM AppWizard

As with normal MFC projects, the first step in creating a new ATL project is to choose **File, New**, click the Projects tab on the New dialog box, and choose “ATL COM Wizard” from the list of projects. Fill in your project’s name, and then click OK to start the AppWizard. Refer to Figure 30.1, which shows the AppWizard dialog box.

---

### Note:

It’s tempting to give the project the same name as the control, but later you’ll need to insert the actual ATL controls. To avoid confusing either yourself or the compiler, name the project by appending `Control` to the control name.

---

The ATL COM AppWizard has only one step, as shown in Figure 30.4. The Server Type group has three choices that govern what type of project you’ll end up with:

- **D**ynamic Link Library—specifies that you want a COM DLL that can act as an in-process server.
- **E**xecutable—builds an .EXE file that can act as an out-of-process server.
- **S**ervice—builds an executable that can run as a Windows NT service. NT services run as background processes, even when no one’s logged in. A service control can process events and data requests from remote sources only.



**FIG. 30.4** The ATL COM AppWizard starts your ATL project; you’ll still have to insert the actual ATL objects.

The Allow merging of proxy/stub files check box controls whether the marshaler for your control is merged into the control’s DLL or is kept in its own DLL. Finally, the Support **M**FC check box controls whether or not the AppWizard will include header files and links to MFC; if you’re not using MFC routines in your control, leave the box unchecked.

Click Finish when you’ve set the options appropriately (the defaults of **D**ynamic Link Library, no MFC support, and no proxy merging are usually correct for most projects.) VC++ displays a confirmation dialog box; when you click its OK button, the AppWizard generates the required C++ class definitions and implementations, plus an IDL file and a proxy/stub makefile, for your project.

## Using the ATL Object Wizard

As soon as you’re done with the AppWizard, you’ll have a nice, but useless, skeleton. Although the files that the AppWizard generates are necessary, they

don't *do* anything by themselves. You still need to actually insert a new ATL control into your project by choosing the New ATL Object command from the Insert menu. This command invokes the ATL Object Wizard, pictured earlier in Figure 30.2. This Wizard enables you to tell VC++ what type of object you want to create. The left-hand pane lists the three types of ATL objects the Wizard supports:

- Ordinary COM objects are just that—they can be used by programs that understand their custom interfaces, but they don't support the full range of ActiveX interfaces
- Full-blown ActiveX controls (usable in any container), Internet Explorer controls (like ActiveX controls minus the non-Internet Explorer COM interfaces), or property pages (used to set the properties of other controls or objects)
- Miscellaneous

After you select the type of object you want to create, you'll see the ATL Object Wizard Properties dialog box, as shown in Figure 30.5. The Names tab enables you to name your new object's C++ class, its COM interface and COM generator (CoClass), and its implementation files. One nice feature of VC++ 5.0 is that you can use long file names when naming files in the Wizards, and this one is no exception.



**FIG. 30.5** The Names tab of the ATL Object Wizard Properties dialog box enables you to specify the class and file names for your new object.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH  
ITKNOWLEDGE](#)[Brief](#) [Full](#)

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

[BROWSE  
BY TOPIC](#)

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The Add control based on combo box enables you to start your control as a subclass of one of the standard Windows controls, enabling you, for example, to build an ATL control that looks like a combo box when it's embedded in a container.

Finally, the Other group holds three settings that didn't belong anywhere else. Normalize DC tells ATL that you want the device context (DC) used for your control to be normalized each time it's used. Although this ensures consistent drawing, it imposes a small performance penalty. The Insertable check box, when enabled, makes your control appear in the standard Insert Object dialog used in OLE-compatible programs; this makes it possible for users to insert your control in almost any OLE container. The Windowed Only checkbox tells ATL whether or not you want the control to be drawn *sans* window in containers that support it; as you'll see in Chapter 32, windowless drawing isn't supported in all containers.

**Setting the Object's Stock Properties.** Every individual control can have its own custom properties; Windows, however, defines a set of *stock properties* that are common to all controls. This standardization provides a consistent way for any control to enable users to customize some aspects of its appearance and behavior.

When creating a new control with the ATL Object Wizard, you can define which stock properties you want to support by specifying them in the Stock Properties tab. As you can see in Figure 30.8, the tab has two lists: one for stock properties supported by the control, and one for unsupported properties. To enable support for a particular property, use the buttons between the two columns to move the desired property into the Supported column. The Object Wizard automatically generates the required code and IDL to use the property.



**FIG. 30.8** The Stock Properties tab enables you to quickly add support for common properties.

- See “Using Stock Properties” for more information on what stock properties are and how to use them, **p. 649**

## Generating Code with the Object Wizard

Like the MFC AppWizard and ClassWizard, the ATL Object Wizard generates some code for you. When you insert a new control into your project, the Object Wizard generates a number of files automatically. (In the sections that follow, *classname* is replaced by whatever you named your class in the Names tab.)

The listings in this chapter are excerpts from the complete source code for the ATLControlWin control. The source is located in the Chap31\ATLControlWin folder on the book's CD-ROM.

**The Control Class Definition.** *classname.h* contains the class definition for your ATL control. The definition makes heavy use of C++ multiple inheritance. A COM class that implements or



uses an interface does so by inheriting from a class representing that interface. Because ATL-based classes implement many interfaces, they inherit from many classes. Listing 30.2 shows all the classes that ATLControlWin's main class inherits from.

**Listing 30.2 ATLCONTROLWIN.H— CATLControlWin Inherits from Many Other ATL and COM Classes**

---

```
class ATL_NO_VTABLE CATLControlWin :
    public CComObjectRootEx<CComObjectThreadModel>,
    public CComCoClass<CATLControlWin, &CLSID_ATLControlWin>,
    public CComControl<CATLControlWin>,
    public IDispatchImpl<IATLControlWin, &IID_IATLControlWin,
        &LIBID_ATLCONTROLLib>,
    public IProvideClassInfo2Impl<&CLSID_ATLControlWin,
        &DIID__DATLControlWin, &LIBID_ATLCONTROLLib>,
    public IPersistStreamInitImpl<CATLControlWin>,
    public IPersistStorageImpl<CATLControlWin>,
    public IQuickActivateImpl<CATLControlWin>,
    public IOleControlImpl<CATLControlWin>,
    public IOleObjectImpl<CATLControlWin>,
    public IOleInPlaceActiveObjectImpl<CATLControlWin>,
    public IViewObjectExImpl<CATLControlWin>,
    public IOleInPlaceObjectWindowlessImpl<CATLControlWin>,
    public IDataObjectImpl<CATLControlWin>,
    public ISupportErrorInfo,
    public IConnectionPointContainerImpl<CATLControlWin>,
    public ISpecifyPropertyPagesImpl<CATLControlWin>,
    public CProxy_DATLControlWin<CATLControlWin>,
    public IPerPropertyBrowsingImpl<CATLControlWin>,
    public IEnumFORMATETC,
    public IDropSource,
    public IDropTarget
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

## Chapter 31

# Creating ActiveX Controls with ATL

- Using the ATL ActiveX Control AppWizard
- Adding methods and properties
- Adding events
- Supporting persistence
- Drawing the ATL control
- Registering the ATL control

Microsoft has positioned ActiveX as a one-size-fits-all solution for application and control developers. Unfortunately, writing ActiveX controls has been a difficult process that requires a great deal of specialized knowledge. One of the primary benefits of the Active Template Library (ATL) is that it greatly eases the pain of writing ActiveX code.

In Chapter 29, “Creating ActiveX Controls,” you learned how to use MFC to write an ActiveX control. In Chapter 30, “Using the ActiveX Template Library,” you learned how to apply ATL in your applications. This chapter combines these two seemingly disparate topics to create ActiveX Controls with ATL.



<http://www.microsoft.com/visualc/prodinfo/atlinst.htm> Remember, this chapter specifically covers ATL 2.1. If you’re using an older version of ATL, you can get the latest version from Microsoft’s Web site. If you’re using Visual C++ 4.0, 4.1, or 4.2, you must also upgrade to version 4.2b or later of Visual C++ to use ATL 2.1.

With the release of Visual C++ 5.0, the ActiveX Template Library (ATL) has matured into a complete ActiveX development framework. The 1.0 and 1.1 releases of ATL introduced ATL to the growing ActiveX development community as an alternative to MFC. These early versions of ATL, however, could only create ActiveX COM objects and ActiveX Automation servers. Although useful in their own right, these capabilities didn’t satisfy developers’ demands for an easier way to write ActiveX controls. ATL version 2.1, the version that ships with VC++ 5.0, now supports the creation of ActiveX controls. Version 1.0 and 1.1 included a single AppWizard for creating basic ATL projects. Version 2.1 includes a number of AppWizards that can be used to create various types of ActiveX components.

In this chapter, you will create an ActiveX control with a complete set of basic methods, properties, and events. The control will also support persistence, as well as understand how to draw its user interface when needed.

## Creating a Basic Control

As with most of the other projects that you've created in earlier chapters, the quickest way to build a new ATL control is to use the Visual C++ AppWizard. Start the Developer Studio development environment, and choose the File, New command. When the New dialog box appears, select the Projects tab (see Figure 31.1), which enables you to define several attributes of the generated application or control; these attributes include the type of application to create, the name of the application, and the location in which you want the project to be created.



**FIG. 31.1** Define the new ATL control project with the New dialog box.

To create an ATL control project, select ATL COM AppWizard in the Type field, and then enter the name that you want to use for the control in the Project Name field. For this project, name it “ATLControl.” The AppWizard will fill in the Location field, but you can change it if needed. After you've set these fields appropriately, click the OK button to start the ATL COM AppWizard.

The next AppWizard step defines the basic architecture of your ATL project (see Figure 31.2). When creating an ActiveX control, you're really creating an in-process DLL server, so you should use the Dynamic Link Library (DLL) option button.

---

**Note:**

An OCX is the same thing as a DLL. Before Microsoft started naming everything in sight “Active,” OLE controls were named with the OCX extension. Because they're just DLLs, though, the OCX suffix was quietly dropped. You can use either extension in your projects.

---

When you're building an ActiveX control, you don't need to use MFC or to support merging the proxy/stub marshaling, so you can leave the Support MFC and Allow merging of proxy/stub code buttons unselected. Click the Finish button to continue.

The New Project Information dialog box is used to confirm the settings that were selected for the project prior to the creation of the actual source files (see Figure 31.3). This step is the last one in the ATL COM AppWizard.



**FIG. 31.2** Define the basic architecture of the ATL COM object with the ATL COM AppWizard.



**FIG. 31.3** Confirm the new project settings with the New Project Information dialog box.

“But wait,” you say. “I haven’t defined any of my control properties.” The ATL COM AppWizard takes a different approach from that of the MFC AppWizard. In MFC, you normally use the AppWizard to customize your application by adding new classes, properties, and methods. For ATL projects, the AppWizard only creates the basic source files, and the remainder of the project is defined by the Object Wizard. The Object Wizard gives you more control of the project implementation because it lets you add any number of ActiveX Controls, Servers, or plain COM objects after the basic project is created. After you confirm your project settings, click the OK button to close the ATL COM AppWizard and create the ATLControl project.

After creating the project, the next step is to add your new ATL control implementation to the project. From the Insert menu, select the New ATL Object menu item. The ATL Object Wizard appears. Select the Controls item in the left panel, and you’ll see a dialog box like the one in Figure 31.4. Your implementation will be a Full Control, so select the Full Control icon. The Object Wizard also enables you to create two other types of components: an Internet Explorer control that supports only the interfaces necessary to be hosted by the Internet Explorer Web browser and a Property Page component, which you’ll need if your control requires property page support. After you’ve selected the appropriate control type, click the Next button to continue.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb’s [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)


**BROWSE**  
 BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

#### Note:

An Internet Explorer control can only be used in Web browsers that support the Internet Explorer control interface. A Full Control, however, can be used in *any* application that supports ActiveX controls, including Internet Explorer. Use IE controls when you want them to run only in Web browsers; otherwise, use full controls.



**FIG. 31.4** Select the type of ATL object to add to your project.

The next dialog box that you will see is the ATL Object Wizard Properties dialog box, which is used to define the specific properties of the new object that you're adding to your project. Select the Names tab, and in the Short Name edit field, type ATLControlWin, as shown in Figure 31.5. The remainder of the edit fields will automatically update to reflect the short name that you added.



**FIG. 31.5** Use the Names tab of the ATL Object Wizard Properties dialog box to name the new control object.

Select the Attributes tab so that you can define the attributes of the control project (see Figure 31.6).

Here's a brief summary of what the most important attributes of the control are:

- The Interface group controls which interfaces your control exposes. ActiveXAutomation controllers need IDispatch interfaces in addition to the normal COM vtable interfaces. Choose Dual (the default) to enable IDispatch interfaces, or choose Custom to only include vtable interfaces.
- The Aggregation group offers options that enable your control's interfaces to be grouped with other controls' interfaces. If you choose Only in the ATL Object Wizard Properties dialog box, your control can

only be used as part of another control.

- The Support ISupportErrorInfo check box specifies whether or not you want to support OLE's ISupportErrorInfo interface. This interface enables clients to receive readable text error messages instead of error codes.
- ActiveX objects receive events via the IConnectionPointXXX classes. The Support Connection Points check box controls whether or not your control can use events; when checked, your control will derive from IConnectionPointXXX so that it can accept events from its container.



**FIG. 31.6** Define the attributes of the new control object with the Attributes tab of the ATL Object Wizard Properties dialog box.

For this control, make sure to check the Support ISupportErrorInfo and Support Connection Points check boxes so your control can return rich error messages and handle events. Leave the other settings alone for now.

You use the Miscellaneous tab to define how the control will draw and act while contained and whether or not your control implementation subclasses a built-in Windows control (see Figure 31.7). For your implementation, you want the control to always create a window whether or not the container is capable of supporting windowless controls, so check the Windowed Only check box. Leave the remainder of the controls at their default settings.

- **See** “Setting the Object’s COM Attributes” for more information about the Threading Model group (and the Free Threaded Marshaller check box), **p. 621**

The Stock Properties tab is used to specify which stock properties the control will support (see Figure 31.8). We’ll discuss properties in “Using Properties” later in this chapter; for now, leave the Stock Properties tab as is, and click the OK button to create the new control object.



**FIG. 31.7** The Miscellaneous tab is used to define some of the basic control behaviors, including whether or not the control is subclassed from a Windows control.



**FIG. 31.8** The Stock Properties tab is used to define the stock properties that

the control object will be created with.

Use the Build menu's Build ATLControl.dll command to compile your control. VC++ will compile the control and register it for you, as described in the next section.

---

**Tip:**

It might seem like Visual C++ is taking an extremely long time to build your ATL control. Because ATL makes heavy use of C++ templates, the compiler has to do quite a bit of extra work, which slows it down. For more information on C++ templates, see "Exploring Templates" in Chapter 21, "Exceptions and Other Power Features."

---

## Registering ATL Controls

ATL provides control registration and unregistration support for you, so you don't have to do anything explicit in your code to support registration. MFC uses a set of constants whose values are used at compile time; in contrast, ATL uses resource information, in the form of a registry script file, to define the information that is added to the registry database. A registry script file is automatically added to the project whenever a new control object is added; one script file is added for each control object.

When you build an ATL control project, the registry script file or files are compiled into the control project as resources. In addition, Visual C++ automatically registers your controls by running REGSVR32 on your registration scripts. The files, which have the extension .rgs, are normal text files that can be edited within the IDE.

---

**Note:**

For more information about the use of registry script files and their syntax, see the Visual C++ books online subject "Registry Scripting Examples—Active Template Library, Articles."

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

• [Advanced](#)[Search](#)• [Search Tips](#)BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Listing 31.1 shows the registry script file for the ATLControlWin control object that you added.

### Listing 31.1 ATLCONTROLWIN.RGS—Registration Script for the ATLControlWin Control

```

{
ATLControlWin.ATLControlWin.1 = s 'ATLControlWin Class'
    {
        CLSID = s '{A19F6964-7884-11D0-BEF3-00400538977D}'
    }
ATLControlWin.ATLControlWin = s 'ATLControlWin Class'
    {
        CurVer = s 'ATLControlWin.ATLControlWin.1'
    }
NoRemove CLSID
    {
        ForceRemove {A19F6964-7884-11D0-BEF3-00400538977D} = s
        0'ATLControlWin Class'
        {
            ProgID = s 'ATLControlWin.ATLControlWin.1'
            VersionIndependentProgID = s 'ATLControlWin.ATLControlWin'
            ForceRemove 'Programmable'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
            ForceRemove 'Control'
            ForceRemove 'Programmable'
            ForceRemove 'Insertable'
            ForceRemove 'ToolboxBitmap32' = s '%MODULE%, 1'
            'MiscStatus' = s '0'
            {
                '1' = s '131473'
            }
            'TypeLib' = s '{A19F6957-7884-11D0-BEF3-00400538977D}'
            'Version' = s '1.0'
        }
    }
}

```

You can also manually register the control with the Tools, Register Control command, or by using the File, Register Controls command in the Active X Control Test Container application, TSTCON32.EXE.

### Testing Your Control with **TSTCON32**

One of the most powerful features of ActiveX controls is that they can be used in any ActiveX-enabled container. Microsoft includes a test container application as part of the Visual C++ suite; it's officially called the ActiveX Control Test Container, and you can find it at DevStudio\VC\bin\TSTCON32.EXE. It will also appear in Visual C++'s Tools menu.



TSTCON32's purpose is to provide a simple, small container for you to use when testing your ActiveX controls. Figure 31.9 shows TSTCON32 with three embedded ATLControl controls in it. You can use TSTCON32 as a container to insert, remove, and inspect ActiveX objects.



**FIG. 31.9** As soon as you've compiled your control, you can test it with the ActiveX Control Test Container application.

Try loading your new control into TSTCON32. Launch the application, and then use the Edit, Inset OLE Control command. When the Insert OLE Control dialog appears, choose "ATLControlWin Class" and click OK. You'll see your control appear as a box that simply says "ATL 2.0." Notice that you can insert many instances of your control—or any other—and drag, remove, and highlight them individually.

---

**Note:**

TSTCON32 is deceptively simple; it actually offers a good simulation of a complex container like Imagineer Technical or Microsoft Word. You can save embedded objects, call individual methods of embedded controls, and view logs of events sent to controls—and that's just for starters! Check out TSTCON32's online help for full details on its capabilities.

---

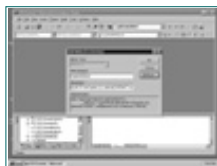
## Creating Methods

Now that you've successfully created, compiled, and tested your basic ActiveX control project, it's time to customize what the control does by adding your own methods to it. As with the MFC classes you've studied in preceding chapters, adding new methods or overriding existing ones is the primary way of getting the desired behavior from objects, and ATL is no exception.

Let's add a method called ShowCaption. The method will accept two parameters: a string that the control will display within its client area and a flag indicating whether the caption should be left-, right-, or center-aligned. We'll make the second parameter optional to illustrate how you can mix required and optional parameters in method calls.

### Adding Methods to the Control

To add methods to an ATL control, you don't use the familiar MFC ClassWizard. Instead, you right-click the class that you want to extend and use the Add Method menu item. From the ClassView tab in the Project Workspace window, select the IATLControlWin interface, click the right mouse button, and select the Add Method menu item. The Add Method to Interface dialog box shown in Figure 31.10 appears.



**FIG. 31.10** Define the ShowCaption method with the Add Method to Interface dialog box.

In the Add Method to Interface dialog box, add the Method Name, ShowCaption, and the Parameters, [in] BSTR bstrCaption, [in, optional] VARIANT varAlignment, [out, retval] long \* lRetVal. The Atttributes button displays a dialog box for adding Interface Definition Language (IDL) attributes for the entire function declaration.

---

**Note:**

All optional parameters must be of type VARIANT, and they must fall at the end of the parameter list. Optional parameters are not managed in any way by OLE. It is the control's responsibility to determine whether or not the VARIANT parameter passed to the method contains data and whether or not to do the following:

- If legal data were passed, use the data passed to the method or convert the data to a useful type, if possible.
- If invalid data were passed, ignore the parameter and use the default value if appropriate.

- Inform the user of an error condition.
- 

You've also added the direction that the parameters flow in the form of [in] and [out] parameter attributes. See Table 31.1 for a complete description of the possible attributes that can be used.

Parameter attributes are used to aid development tools in determining how parameters are used within a function call. A tool like Visual Basic will hide the details of how parameters are handled—such as creating and destroying memory—based on these and other attributes in the type library. This is why the type library is so important to ActiveX component development.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief

Full

● [Advanced](#)

● [Search](#)

● [Search Tips](#)

BROWSE

BY TOPIC

[an error occurred while processing this directive]

Note that the IDL parameter attributes are added directly to the parameter list. If you're a die-hard IDL hacker, you can add your own attributes, but you don't have to know IDL to use Click OK to add the method to the control.

**Table 31.1 Parameter Flow Attributes**

Direction	Description
in	Parameter is passed from control to container.
out	Parameter is returned from container to control.
in, out	Parameter is passed from control to container, and the container returns a parameter.
out, retval	Parameter is the return value of the method and is returned from the container to the control.

To complete ShowCaption's implementation, you need to add an enumeration that specifies all the valid alignment settings (see Listing 31.2), as well as two member variables: m\_lptstrCaption stores the caption string, and m\_lAlignment holds the alignment setting. Listing 31.3 shows the relevant portion of the CATLControlWin class definition.

The complete source code and executable file for the ATLControlWin control is located in the Chap31\ATLControlWin folder on the book's CD-ROM.

**Listing 31.2 ALIGNMENTENUMS.H—Alignment Enumeration Include File**

```
#if !defined _ALIGNMENTENUMS_H
#define _ALIGNMENTENUMS_H
// caption alignment enumeration
typedef enum tagAlignmentEnum
{
    EALIGN_LEFT = 0,
    EALIGN_CENTER = 1,
    EALIGN_RIGHT = 2,
}EALIGNMENT;
#define EALIGN_LEFT_TEXT    "Left"
#define EALIGN_CENTER_TEXT  "Center"
#define EALIGN_RIGHT_TEXT   "Right"
#endif // #if !defined _ALIGNMENTENUMS_H
```

**Tip:**

The EALIGNMENT enumeration is defined in its own stand-alone include file so that you can use it elsewhere. It's good practice to separate enumerations such as this so that you can easily use them in more than one source file.

**Listing 31.3 ATLCONTROLWIN.H—Alignment Enumeration Definition and Member Variables Added to Class Definition**

```
// ATLControlWin.h : Declaration of the CATLControlWin
#ifndef __ATLCONTROLWIN_H_
#define __ATLCONTROLWIN_H_
#include "resource.h"           // main symbols
```

```
#include "alignmentenums.h"
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CATLControlWin
class ATL_NO_VTABLE CATLControlWin :
. . .
protected:
    // storage variable for the caption
    LPTSTR m_lptstrCaption;
    // storage variable for the alignment
    long m_lAlignment;
};
```

---

Note that `m_lAlignment` is declared as type `long` and not as the `EALIGNMENT` enumeration type. ActiveX Automation restricts the types of data that can be passed to methods; only data types that can be passed in a `VARIANT` can be used in methods and properties. By declaring the `m_lAlignment` member as `long`, you avoid having to explicitly convert the value by casting to the enumerated type when it is retrieved from the `VARIANT` parameter in the `ShowCaption` method.

Of course, you'll also need to initialize your member variables in the control's constructor, as shown in Listing 31.4.

#### Listing 31.4 ATLCONTROLWIN.H—Member Variable Initialization

---

```
. . .
public IConnectionPointContainerImpl<CATLControlWin>,
    public ISpecifyPropertyPagesImpl<CATLControlWin>
{
public:
    CATLControlWin()
    {
        // NULL terminate the string reference
        m_lptstrCaption = new TCHAR[1];
        m_lptstrCaption[0] = '\\0';
        // set the alignment to the default of left
        m_lAlignment = EALIGN_LEFT;
    }
DECLARE_REGISTRY_RESOURCEID(IDR_ATLCONTROLWIN)
. . .
```

---

### The *ShowCaption* Implementation

Just like the MFC AppWizard, the ATL Object Wizard only generates a stub implementation for methods you add; you must still write the code that makes them work. The `ShowCaption` method contains all of the code for setting the caption and the alignment style, and it demonstrates how to do data type conversions and what to do with optional parameters. Let's examine it section by section, starting with the code in Listing 31.5.

#### Listing 31.5 ATLCONTROLWIN.CPP—Receiving and Converting Parameters for the Method

---

```
STDMETHODIMP CATLControlWin::ShowCaption (BSTR bstrCaption, VARIANT
OvarAlignment, long * lRetVal)
{
    USES_CONVERSION;        // needed for the W2A macro
    HRESULT hResult = S_OK;

    // initialize our return value
    *lRetVal = FALSE;

    // convert the input string to ANSI: OLE provides us a BSTR,
    // but we need an ordinary null-terminated string instead.
    LPTSTR lptstrTempCaption = W2A(bstrCaption);
```

---

ShowCaption can be called in two different ways: directly through IATLControlWin's custom COM interface or via the IDispatch implementation used by ActiveX Automation. Supporting both interfaces requires a few changes to the method's implementation as compared to a similar MFC method. First, the function is declared as STDMETHODCALLTYPE, which expands to an HRESULT return type. The return value is used by OLE to determine whether or not the method call succeeded.

The caption parameter is passed in differently, too. Instead of a CString or some form of null-terminated string, OLE passes all strings with Unicode encoding. MFC hides the implementation details concerning how the strings are managed; the developer simply uses the appropriate string data type based on the target application and platform. Because ATLControlWin doesn't use MFC, it must convert the string to ANSI. Note the use of the USES\_CONVERSION and W2A macros to convert the string from UNICODE to ANSI.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

If this were an ordinary MFC-based control or ActiveX control, the code would call the derived InvalidateControl function to force the control to repaint itself. Because ATL-based controls use OLE events to trigger actions, however, this code instead calls the FireViewChange method to trigger the redraw. We'll discuss events more in the "Responding to OLE Events" section, but first we need to add property support to ATLControlWin.

## Using Properties

Methods encapsulate what a control can do (draw itself, play a sound, load an URL, and so on). **Properties** encapsulate the data held by the control: its foreground and background colors, the name of a .WAV file, or an URL. ActiveX control properties fall into three categories: user-defined, stock, or ambient.

- **User-defined** properties make your control unique; they are implementation-specific and have meaning only to the component that contains them. User-defined properties can be further broken into those properties that are defined only as their specific data type (**normal** properties) and those with additional parameters (**parameterized** properties).
- **Stock** properties are predefined properties whose basic meaning is the same for all controls. For example, background and foreground color are both stock properties. However, stock properties are not implemented by default; you still have to implement support for them just like user-defined properties. They are predefined only to imply a certain level of uniformity between various control implementations.
- By contrast to stock properties, **ambient** properties are standard properties that are supported **by the container** to provide a default value to the control that uses them. When a user adjusts an ambient property, it applies to controls in the container.

## Creating Normal User-Defined Properties

A **normal** property is a property that is declared as a single type, for example, long or BSTR. Normal properties have no parameters. Because the control's ShowCaption method will have a member variable to track the text's alignment, we'll expose it as a normal property so that it can be externally controlled.

You add properties to your control in much the same way as methods. From the ClassView tab in the Project Workspace window, select the IATLControlWin interface, click the right mouse button, and choose the Add Property menu item.

In the Add Property to Interface dialog box, set the Property Type to long and the Property Name to Alignment, and leave the remainder of the settings at their default values (see Figure 31.11). Click OK to confirm the entry and to close the dialog box.



**FIG. 31.11** Define the Alignment property attributes in the Add Property to Interface dialog box.

When you add a property in this way, the Object Wizard adds stub routines called **accessors** to your control's class. Accessors usually come in pairs: The get\_XXX accessor enables you to read a property value, and its twin, set\_XXX, lets you change a property's value. (Of course, instead of XXX, you'll use the property name.) The Get Function and Put Function check boxes in the Function Type group of the Add Property to Interface dialog box control whether one, both, or neither accessors are automatically added when you add a property. Accessor definitions are also added to the control's IDL file.

After adding the property, you should also add a dispid constant that can be used from within the control implementation source files. Listing 31.9 shows the typedef that is added to the IDL file. The dispid constants are added as an enumeration so that the MIDL compiler will generate an enumeration in the ATLControl.h header file, which defines the interfaces and classes available in the control IDL file. The reason for adding the dispid as a set of constants is the same reason for having any constant: If the value of the dispid changes, you won't have to search your source code trying to find where you used the value. When you define the dispid constant in the IDL file, you should also change the Alignment property function declarations to use the constant; these changes are also shown in Listing 31.9.

---

**Listing 31.9 ATLCONTROL.IDL—Dispid Enumeration Added to the IDL File to Aid in the Support of Properties in the Control**

---

```
. . .
    typedef enum propdispid
    {
        dispidAlignment = 2,
    } PROPDISPIDS;
. . .
    interface IATLControlWin : IDispatch
    {
        [id(1), helpstring("method ShowCaption")]
        HRESULT ShowCaption([in] BSTR bstrCaption,
            [in, optional] VARIANT varAlignment, [out, retval] long *
            lRetVal);
        [propget, id(dispidAlignment), helpstring("property Alignment")]
        HRESULT Alignment([out, retval] long *pVal);
        [propput, id(dispidAlignment), helpstring("property Alignment")]
        HRESULT Alignment([in] long newVal);
    };
. . .
```

---

Now that you've modified the IDL definitions, it's time to flesh out the `get_Alignment`/`put_Alignment` accessors. To get to these functions, open the `ATLControlWin.cpp` file, or open the functions by double-clicking them in the ClassView tab of the Project Workspace window.

Both of these functions use the `m_lAlignment` member variable that you added earlier. `get_Alignment`, as shown in Listing 31.10, is simple: It returns `m_lAlignment`'s value in the output parameter.

---

**Listing 31.10 ATLCONTROLWIN.CPP—The `get_Alignment` Property Accessor**

---

```
STDMETHODIMP CATLControlWin::get_Alignment(long * pVal)
{
    HRESULT hResult = S_OK;
    // return our current setting
    *pVal = m_lAlignment;
    return hResult;
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The `set_Alignment` function is simple too, but more complicated than `get_Alignment`. As shown in Listing 31.11, `set_Alignment` first tests the new alignment; if it's out of range, it returns `S_FALSE` to the caller and doesn't change anything. If the value is within limits, it updates `m_lAlignment` and notifies the control of the change by calling `SetDirty`.

### Listing 31.11 ATLCONTROLWIN.CPP—The `set_Alignment` Property Accessor

```

STDMETHODIMP CATLControlWin::put_Alignment(long newVal)
{
    // Don't do anything if the supplied value is out of range
    if (newVal < EALIGN_LEFT || newVal > EALIGN_RIGHT)
        return S_FALSE;

    m_lAlignment = newVal;

    // tell the control that its property's changed
    this->SetDirty(TRUE);

    // update the browser for this property
    this->FireOnChanged(dispidAlignment);

    // redraw the control
    this->FireViewChange();

    return S_OK;
}
    
```

Next, `FireOnChanged` has the effect of forcing the container to refresh its property browser to reflect the new value. This step is very important because the value of the property could change without the container's knowledge, either through the control's property sheet or, in some cases, in response to another function call.

You might be asking "Why didn't I add `FireOnChanged` to the `ShowCaption` method?" Well, you could have, but it wouldn't do much—`ShowCaption` will never be executed while the control is used in design mode, which is the purpose of `FireOnChanged`. The last thing that the `set_Alignment` method does is invalidate the control's drawing area so that it will redraw using the new information.

## Creating Parameterized User-Defined Properties

A *parameterized* property is a property that, in addition to being of a specific type (for example, `BSTR` or `long`), accepts one or more additional parameters to further define the data of the property. Parameterized properties can be useful for properties that represent collections of data where the additional parameter is the index into the collection.

You are going to expose the control's `m_lptstrCaption` member variable as a parameterized property, in addition to the `ShowCaption` method. Parameterized properties are added in the same manner as normal properties:

1. From the ClassView tab in the Project Workspace window, select the `IATLControlWin` interface.
2. Click the right mouse button, and select the `Add Property` menu item.
3. In the Add Property to Interface dialog box, set the Property Type to `BSTR` and the Property Name to `CaptionProp`.
4. Add the Parameters string **[in, optional] VARIANT varAlignment**, and leave the remainder of the settings at their default values (see Figure 31.12).
5. Click OK to confirm the entry, and then close the dialog box.

#### Note:

Even though the `VARIANT varAlignment` is defined as [optional] for both the `get_CaptionProp` and `put_CaptionProp` functions, only the `get_CaptionProp` implementation is truly optional. The alignment parameter has to be added as optional because the ATL ClassWizard won't allow you to generate two separate functions that have the same name and ID. The [optional] attribute can be removed from the `put_CaptionProp` function later without any adverse effect.





**FIG. 31.12** Define the Caption property propget function attributes.

As with the normal user-defined properties shown earlier in Listing 31.9, you should add a dispid constant to your IDL file, as shown in Listing 31.12. Don't forget to update the accessor definitions in the IDL file to use dispidCaptionProp instead of the existing id.

---

**Listing 31.12 ATLCONTROL.IDL—Adding a New dispid Constant for the CaptionProp Property**

---

```
typedef enum propdispid
{
    dispidAlignment = 2,
    dispidCaption = 3
}PROPDISPIDS;
```

The `get_CaptionProp` method, as shown in Listing 31.13, returns data from the property to the caller. Your implementation should ignore the alignment parameter because it is of no use to you in this context; you simply return the `m_lpszCaption` member. If the return value, `BSTR * pVal`, passed to the `get_CaptionProp` function already points to another string, it must be freed. To return the requested value, `get_CaptionProp` uses the `SysAllocString` function to create a new BSTR. `SysAllocString` expects an OLE string of type `OLECHAR`, so it's necessary to convert the ANSI string to an `OLECHAR` string with the `T2OLE` macro and then allocate a BSTR from that.

---

**Listing 31.13 ATLCONTROLWIN.CPP—The `get_CaptionProp` Accessor**

---

```
STDMETHODIMP CATLControlWin::get_CaptionProp(VARIANT varAlignment, BSTR * pVal)
{
    USES_CONVERSION; // needed for the T2OLE macro
    // if there is a string already, free it because we are going to replace
    it
    if(*pVal)
        ::SysFreeString(*pVal);

    // convert the ANSI string to an OLECHAR and then allocate a BSTR
    *pVal = ::SysAllocString(T2OLE(m_lpszCaption));
    return S_OK;
}
```

`put_CaptionProp` simply defers to the `ShowCaption` implementation because `ShowCaption` already does everything that you need. This delegation enables you to reuse `ShowCaption` simply by wrapping it in an accessor that external callers can use. See Listing 31.14 to see how the delegation is implemented.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Listing 31.14 ATLCONTROLWIN.CPP—The put\_CaptionProp Delegates Its Work to ShowCaption

```

STDMETHODIMP CATLControlWin::put_CaptionProp(VARIANT varAlignment, BSTR newVal)
{
    long lRetVal=0;
    // defer to the ShowCaption implementation
    HRESULT hResult = this->ShowCaption (newVal, varAlignment, &lRetVal);

    // if the function returned success, notify the control
    if(TRUE == lRetVal)
        this->SetDirty(TRUE);
    return hResult;
}
  
```

## Using Stock Properties

A **stock** property is a property that is understood by a control and its container and that has a predefined meaning to both. Stock properties are intended to provide basic, uniform functionality to all the controls and containers that implement them. Stock properties do not require you to implement a lot of code; you just hook into the existing property.

Stock properties can be added to a control in two ways. The first way is by using the ATL Object Wizard during the actual creation of the control. You might recall that earlier in the chapter, one of the options in the ATL Object Wizard Properties dialog box is the Stock Properties tab (see Figure 31.8 to refresh your memory). If you add any stock properties at this point, the ATL Object Wizard will add the class CStockPropImpl<...> to your class declaration and will add the necessary IDL get/put\_function declarations for each one of the properties.

The Object Wizard will also add a member variable to your class for each one of the properties. The CStockPropImpl<...> template class contains declarations for all of the available stock properties in the form of IMPLEMENT\_STOCKPROP and IMPLEMENT\_BSTR\_STOCKPROP macros. The macros define all of the appropriate get put\_function implementations for you; you need to use only the member variable when you want to use the stock property. (If you want to, you could do this manually by adding the CStockPropImpl<...> class after your control object has been created. Note that the CStockPropImpl<...> class replaces the IDispatchImpl<...> class.)

#### Note:

The documentation about the use of the IMPLEMENT\_STOCKPROP and IMPLEMENT\_BSTR\_STOCKPROP macros within your class is somewhat misleading. The macros depend on the CStockPropImpl<...> class, and they cannot simply be added to your control implementation, as is implied by the ATL documentation on support of stock properties.

The second method of adding a stock property is the same method as for user-defined properties:

1. First, from the ClassView tab in the Project Workspace window, select the IATLControlWin class.
2. Click the right mouse button, and select the Add Property menu item.
3. In the Add Property to Interface dialog box, set the Property Type to OLE\_COLOR and the Property Name to BackColor, and leave the remainder of the settings at their default values (see Figure 31.13).
4. Click OK to confirm the entry and close the dialog box.



**FIG. 31.13** Add the BackColor stock property to the control with the ATL Add Property to Interface dialog box.

Next, you must modify the IDL file to reflect the correct dispid for the BackColor property. Because it's a stock property, you need to replace the existing ID value with DISPID\_BACKCOLOR. (If you add stock properties with the Stock Properties tab of the ATL Object Wizard, the correct dispsids will be automatically inserted.)

After updating the IDL file, you should add a member variable (which we'll call m\_BackColor) to your class declaration so that you can store the BackColor property's value. Make sure to add code in the class constructor to initialize m\_BackColor. The final step is to update the automatically generated stubs of the get\_BackColor and put\_BackColor functions to return and store the BackColor property (see Listing 31.15).

---

**Listing 31.15 ATLCONTROLWIN.CPP—The get\_BackColor and put\_BackColor Accessors**

---

```
STDMETHODIMP CATLControlWin::get_BackColor(OLE_COLOR *pVal)
{
    // return the color
    *pVal = m_BackColor;
    return S_OK;
}

STDMETHODIMP CATLControlWin::put_BackColor(OLE_COLOR newVal)
{
    // Don't do anything if the supplied value is the same as what we
    // already have
    if(newVal != m_BackColor)
    {
        m_BackColor = newVal;

        // tell the control that its property's changed
        this->SetDirty(TRUE);

        // update the browser for this property
        this->FireOnChanged(dispidAlignment);

        // redraw the control
        this->FireViewChange();
    }
    return S_OK;
}
```

---

No matter how you added the BackColor stock property to your class, you can access it in your class through the member variable, just as you do any other property.

## Using Ambient Properties

**Ambient** properties are properties implemented in the container in which the control resides, as opposed to stock properties, which are implemented in the control and not the container. Ambient properties share the same set of predefined meanings and dispsids as those of stock properties. To use an ambient property, the control need request only the property value from the container and apply it in whatever manner is appropriate for the property type. The use of ambient properties enables the control to conform to the same settings as those of the container in which it resides. This procedure provides much better integration between the control and its container.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

Brief    Full  
 + [Advanced](#)  
[Search](#)  
 + [Search Tips](#)

 BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Take the previous example of adding the BackColor stock property to the sample control implementation. Defined as a stock property, the control's user can change the background color of the control or leave it as is. If the color is different from that of the container or if the container's background color changes for some reason, the colors won't match and will give the appearance of a poorly integrated and written application. However, if the control simply uses the ambient background color of its container, the control's background will always match that of the container. The specific requirements of your control implementation will determine how much ambient property support you provide in your control.

When you create a Full Control or Internet Explorer control, your control is automatically subclassed from CComControl. The CComControl class family provides accessors for all of the standard ambient properties. For example, GetAmbientBackColor() returns the ambient background color, while GetAmbientTextAlign() returns the ambient text alignment setting. Because ambient properties come from the container, you can't set them from your control, and CComControl doesn't provide any set functions.

## Creating Property Sheets

Property sheets give controls a way to display their properties for review and editing, usually using a tabbed-dialog interface. The original intent of property sheets was to provide a way to set control properties when the control container did not support property browsing facilities. Although property sheets are still common and are frequently used, they are probably not necessary for all implementations. Your specific requirements will determine whether or not your control should contain a property sheet. The official OC96 line is that all controls should have property sheets; this is good advice for commercially developed and distributed controls, but your specific control might not need or benefit from their use. The majority of development environments—especially Visual Basic 5.0 and Delphi 2.x—already have excellent property-browsing facilities.

**Adding the Property Sheet Object.** Before you can implement the property sheet, you must first add a property sheet object to your control. This is done through the ATL Object Wizard. From the Insert menu, select the New ATL Object menu item. Within the ATL Object Wizard, select the Controls item in the left panel to display the types of control components that can be added (see Figure 31.14). Your implementation will be a Property Page, so select the Property Page icon. Click the Next button to continue.



**FIG. 31.14** Add a property page object to your class with the ATL Object Wizard.

Select the **Names** tab in the ATL Object Wizard Properties dialog box, which you saw earlier in Figure 31.5, and in the **Short Name** edit field, type `ATLControlWinPPG`; the remainder of the edit fields will automatically update, reflecting the short name that you added. The other fields can be changed, but in this case, you can use the default values.

Next, select the **Strings** tab, and enter the string `General` to the **Title** edit field and `ATLControlWin Property Page` to the **Doc String** edit fields. The **Helpfile** edit field is where you add the name of the help file associated with the property page; for now, leave the value at its default setting.

Click **OK** to add the property page object to your control. The ATL Object Wizard will add the necessary code to your control implementation to use the new property page object, including changing the `OBJECT_MAP`, the IDL file, and all of the registry and resource information.

Because property sheets are tabbed dialog boxes, most of your work on the property sheets will be done with the dialog editor. Select the **Resource View** in the **Project Workspace** window. From the list of dialogs, select `IDD_ATLCONTROLWINPPG`, and double-click the entry to open the resource editor.

Using the resource editor, remove the static text control with the caption `Insert Your Controls Here`, and place a static text control and a combo box on the dialog box.

Using the mouse, select the label control on the form, and click the right mouse button. In the menu that appears, choose the **Properties** menu item. On the **General** tab, set the **ID** of the control to `IDC_ALIGNMENTLABEL`, and set the **Caption** to `Alignment`, as shown in Figure 31.15. Select the **Styles** tab, and set the **Align Text** property to `Right`. Close the dialog box to save the information.



**FIG. 31.15** Add the controls to the Property Sheet dialog box that will display your property data.

Now, right-click the combo box control, and in the menu that appears, select the **Properties** menu item. On the **General** tab, set the **ID** of the control to `IDC_ALIGNMENTCOMBO`. On the **Styles** tab, set the **Type** to `Dropdown`, and uncheck the **Sort** check box. Close the dialog box to save the information.

The next step is to add some code to the control to make it use the dialog resources that you just created. Close the resource editor, and open the ATLControlWinPPG.h header file. Your implementation of the property page requires several resource definitions and the Alignment enumeration that you created earlier. The values are added to the property page header file in the form of include statements (see Listing 31.16.)

**Listing 31.16 ATLCONTROLWINPPG.H—Adding Include Statements to Reference the Alignment Enumerator and the Main Control Definition**

---

```
. . .  
#include "resource.h"           // main symbols  
#include "alignmentenums.h"  
#include "ATLControl.h"  
EXTERN_C const CLSID CLSID_ATLControlWinPPG;  
. . .
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Because more than one object can have a reference to the property page, Apply loops through all of its references, checking to see whether or not each implements the IATLControlWin custom COM interface. If it finds an interface that it can use, you can call the appropriate member functions for each of the properties that you support. If, for some reason, the setting of the property causes an error, the property page notifies the user with an error message, resets the m\_bDirty flag, and exits the function. At this point, the property dialog box closes, and control returns to the development environment hosting the control.

**Linking the Page and Control Together.** The property page implementation is complete; however, the control is still not aware of the property page's existence and the fact that the control and the property page are related. To connect the property page to the control, you must add an entry to the BEGIN\_PROPERTY\_MAP macro in the class definition of the control (see Listing 31.21). The macro PROP\_ENTRY is used in place of the PROP\_PAGE macro so that the property will be automatically made persistent when the control is saved.

### Listing 31.21 ATLCONTROLWIN.H—Adding an Additional Entry to Take Advantage of the Property Page

```

BEGIN_PROPERTY_MAP(CATLControlWin)
    // PROP_ENTRY("Description", dispid, clsid)
    PROP_ENTRY("Alignment", dispidAlignment, CLSID_ATLControlWinPPG)
    PROP_PAGE(CLSID_StockColorPage)
END_PROPERTY_MAP( )
    
```

## Responding to OLE Events

Properties and methods are a way for a programmer to communicate with a control from within the container application. **Events** are a way for the control to communicate with the container. For ActiveX controls, events are nothing more than IDispatch interfaces that are implemented on the container side of the container/control relationship. The mechanism that events are based on is known as a **connection point**. A connection point is simply a description of the type of interface that is required to communicate with the container. Connection points are not restricted to only IDispatch interfaces; rather, they can be of any COM interface that is understood by both components. For that matter, connection points/events are not restricted only to controls; they can be used in any COM implementation. Controls were simply the first to take advantage of them. For more information regarding connection points, refer to the documentation in the OLE online help or to Kraig Brockschmidt's *Inside OLE*, 2nd Edition.

### Adding the Event Interface

Because no ClassWizards are available to aid you, adding events in ATL requires a bit more work than is required in MFC. The first step is to add the event interface to your IDL file, as shown in Listing 31.22. This COM interface makes it possible for your control to receive events sent by other controls or its container.

### Listing 31.22 ATLCONTROL.IDL—Manually Adding Event Interfaces to Your Control's IDL



```
[
    uuid(C31D4C71-7AD7-11d0-BEF6-00400538977D),
    helpstring("ATLControlWin Event Interface")
]

dispinterface _DATLControlWin
{
    properties:
    methods:
        [id(1)] void Change([in, out]BSTR * bstrCaption,
[in, out] long * lAlignment);
};

[
    uuid(A19F6964-7884-11D0-BEF3-00400538977D),
    helpstring("ATLControlWin Class")
]
coclass ATLControlWin
{
    [default] interface IATLControlWin;
    [default, source] dispinterface _DATLControlWin;
};
```

---

---

**Caution:**

Remember to use the guidgen.exe program to create a new UUID for your IDispatch interface. Do not reuse the UUIDs in the sample, because they might collide with other registered UUIDs on your machine.

---

## Defining an Event Method

In addition to the event interface, you need to define an event method. For this control, we'll add the Change method. Any event methods that you add must also be added to the event interface. Change has two parameters: The first parameter is a string called bstrCaption, passed by reference (BSTR \*). The second is a long called lAlignment, also passed by reference (long \*). You are passing the parameters by reference to give the container application the opportunity to change the values if necessary. You must also add the event interface to the CoClass interface for the control as the default, source interface; the code in Listing 31.22 contains these changes.

Event interfaces are based on connection point interfaces, IConnectionPoint and IConnectionPointContainer. You must add these interfaces to your control implementation. Doing so by hand is tricky; to make it easier to add interfaces, ATL provides a proxy generator. To generate the proxy code, follow these steps:

1. From the **Project** menu, choose the **Add to Project** menu item, and then choose the **Components and Controls** submenu item.
2. In the **Components and Controls Gallery** dialog box, double-click the **Developer Studio Components** folder.
3. After the **Components and Controls Gallery** dialog box is refreshed with data, double-click the **ATL Proxy Generator** icon (see Figure 31.16).





**FIG. 31.16** Select the ATL Proxy Generator from the Developer Studio Components.

4. Click OK to confirm that you want to insert the ProxyGen component.
5. When the ATL Proxy Generator dialog box appears, click the & button to display the Open dialog box, and then select the ATLControl.tlb file and click Open.
6. Choose the DATLControlWin entry in the Not Selected list box, and click the > button to move the entry to the Selectd list box (see Figure 31.17). Ensure that the Proxy Type is set to Connection Point, and click Insert.



**FIG. 31.17** Select the DATLControlWin class in the ATL Proxy Generator dialog box.

7. A Save dialog appears with the file CPATLControl.h in the File name edit box. Click Save to save the generated file, then click OK in the confirmation dialog box that indicates that the operation was successful.
8. Close the ATL Proxy Generator and Components and Controls Gallery dialogs.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

The CPATLControl.h file contains the CProxyDATLControlWin class, which implements your event interface. It also provides a stub for the Change event method, in the form of a routine named Fire\_Change. The next step is to add the CProxyDATLControlWin interface to your control implementation. First, add the CPATLControl.h file as an include file to your ATLControlWin.h file:

```
#include "CPATLControl.h"
```

Next, update CATLControlWin's inheritance structure by adding a line to CATLControlWin's class definition. This line makes CATLControlWin inherit from CProxyDATLControlWin as well as the other classes it already inherits from:

```
public CProxyDATLControlWin<CATLControlWin>
```

You also have to add the unique ID of the event interface to the IProvideClassInfo2Impl interface. Because the event interface is IDispatch-based, all you have to do is supply the dispatch interface ID (DIID) in place of the default NULL parameter:

```
public IProvideClassInfo2Impl<&CLSID_ATLControlWin, &DIID__DATLControlWin,
&LIBID_ATLCONTROLLib>
```

---

**Tip:**

See the ATL and ActiveX documentation for information regarding the implementation of the IProvideClassInfo2 interface.

---

Finally, add the event interface to the BEGIN\_CONNECTION\_POINT\_MAP macro in the control's class declaration. This macro is empty by default. Every time you add an event interface, you'll add a corresponding entry here, like this:

```
BEGIN_CONNECTION_POINT_MAP(CATLControlWin)
    CONNECTION_POINT_ENTRY(DIID_DATLControlWin)
END_CONNECTION_POINT_MAP( )
```

Your event interface is now completely implemented. The only remaining step is to add the Fire\_Change method calls wherever appropriate in your control. Because your implementation of the Fire\_Change method enables the user of the control to change the data that is passed to the event, using a universal helper function makes the code easier to maintain. By implementing a simple helper function, FireChange (with no parameters), you can encapsulate the data management associated with the method and its parameters (see Listing 31.23). Remember to add the FireChange function prototype to your class definition in the ATLControlWin.h header file.

---

**Listing 31.23 ATLCONTROLWIN.CPP—Isolating the Details of the Change Mechanism from the Code that Fires It**

---

```
void CATLControlWin::FireChange(void)
{
    // needed for the W2A macro
    USES_CONVERSION;

    // get a BSTR that can be passed via the event
    BSTR bstrCaption = ::SysAllocString(T2OLE(m_lptstrCaption));
```

```

// fire the change event
this->Fire_Change(&bstrCaption, &m_lAlignment);

// convert the string to ANSI
LPTSTR lptstrTempCaption = W2A(bstrCaption);

// free the data that was passed back
::SysFreeString(bstrCaption);

// if we have a string
if(m_lptstrCaption)
{
    // delete the existing string
    delete [] m_lptstrCaption;
    // clear the reference just to be safe
    m_lptstrCaption = NULL;
}
// allocate a new string
m_lptstrCaption = new TCHAR[lstrlen(lptstrTempCaption) + 1];
// assign the string to our member variable
lstrcpy(m_lptstrCaption, lptstrTempCaption);
}

```

---

Finally, your event code is completely implemented. The FireChange function enables you to hide the details involved in implementing the change event from the rest of the program. If you decide to change what the event does in the future, the changes will only be in FireChange, so changes will impact only one function rather than many.

One last thing—the ShowCaption method requires that you fire a Change event if the data changes, so you have to add one line to the code in Listing 31.8 (right before the call to FireViewChange) to do so:

```

this->FireChange( )

```

You need to add a corresponding call to the put\_Alignment function, but not to put\_CaptionProp, which delegates its work to ShowCaption. If you add other new methods to ATLControlWin, you might need to add calls to FireChange() there as well.

## Supporting Persistence

**Persistence** refers to the capability of a component to retain its state across execution lifetimes. In other words, regardless of the number of times that the control is started and stopped, it remembers that you changed its background color from white to mauve (even if it finds the color mauve revolting).

Adding property persistence in ATL is trivial. The only thing that you have to do is add the property to the BEGIN\_PROPERTY\_MAP macro (see Listing 31.24), and you are finished; the rest is up to ATL. You can make user-defined or stock properties persistent, but not ambient properties, because they're supplied by the container.

### Listing 31.24 ATLCONTROLWIN.H—Adding Properties to the Property Map

---

```

BEGIN_PROPERTY_MAP(CATLControlWin)
    PROP_ENTRY("Alignment", dispidAlignment, CLSID_ATLControlWinPPG)
    PROP_ENTRY("BackColor", DISPID_BACKCOLOR, CLSID_ATLControlWinPPG)
    PROP_PAGE(CLSID_CColorPropPage)
END_PROPERTY_MAP( )

```

---

## Drawing the Control

Most controls will have some form of user interface. (However, because the release of the OC 96 specification and ActiveX, UI is no longer a requirement; you can build invisible controls that the user can't see but which can offer services to other controls and programs.)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Chapter 32

# Advanced Programming with ATL

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

- Adding asynchronous properties
- Supporting property enumeration
- Enabling optimized drawing
- Providing Clipboard and Drag-and-Drop support
- Using Advanced ActiveX features in ATL

In preceding chapters, you learned how to use the Active Template Library (ATL) in your applications and ActiveX controls. Now it's time to explore some advanced features of ATL that you can use to extend and enhance your controls' usefulness. In addition to familiar features like Clipboard and Drag- and-Drop support, you will learn how to implement asynchronous properties and optimized drawing, which are key parts of the OLE Control 96 (OC 96) specification.



The complete source code and executable file for the ATLControlWin control is located in the Chap31\ATLControlWin folder on the book's CD-ROM.

## Adding Asynchronous Properties

In Chapter 31, "Creating ActiveX Controls with ATL," you learned how to use ambient, stock, and user-defined properties in your ATL ActiveX controls. In addition to these three types, ATL enables you to use *asynchronous properties* in your controls.

Asynchronous properties are properties in which the data is loaded in a background process or thread. Properties that typically contain a large amount of data, such as bitmapped images or sounds, can be made asynchronous so as not to interfere with the normal processing of the control and the container. When an asynchronous property is loaded, the control can still respond to user interface events, and the user can stop the data loading.

For example, a control that downloads and parses a file of stock quotes from an Internet server can load the data asynchronously. When the control needs to load data, it can begin the download and register a callback function to be called when the transfer is complete. The callback function is called to signal that the property's data has been retrieved and that it can be used normally. Your controls can process the incoming data as it arrives, or they can wait until all data has arrived before doing anything with it.

These callbacks are implemented with OLE's IBindStatusCallback interface and the corresponding CBindStatusCallback class. In addition, your control should support the ReadyState property, which is used by containers to tell what state the control is in. To properly support ReadyState, you also need to add the ReadyStateChange event, which is used by the control to notify the container that the value of the control's ReadyState has changed.

### Adding the ReadyState Property

You add the ReadyState property just like any other property, as described in Chapter 31:

1. From the ClassView tab in the Project Workspace window, select the IATLControlWin interface.
2. Click the right mouse button, and select the Add Property menu item.
3. In the Add Property to Interface dialog box, set the Property Type to long and the Property Name to ReadyState.
4. Uncheck the Put Function check box, and leave the remainder of the settings at their default values (see Figure 32.1).
5. Click OK to confirm the entry and close the dialog box.



**FIG. 32.1** Add the ReadyState property to the control.

ReadyState is a stock property, even though it doesn't appear in the ATL Object Wizard stock property dialog box. Because you added it as a user-defined property, you must change its dispid. Before you proceed, open the ATLControl.idl file, and change the dispid of the ReadyState property to DISPID\_READYSTATE. Listing 32.1 shows the IDL file after editing.

**Listing 32.1 ATLCONTROL.IDL—Changing the dispid of the ReadyState Property to the Stock Property dispid—DISPID\_READYSTATE**

```
. . .
interface IATLControlWin : IDispatch
{
    [id(1), helpstring("method CaptionMethod")] HRESULT CaptionMethod(
        [in] BSTR bstrCaption, [in, optional] VARIANT varAlignment,
        [out, retval] long * lRetVal);
    [propget, id(DISPID_READYSTATE), helpstring("property ReadyState")]
        HRESULT ReadyState([out, retval] long *pVal);
    [propget, id(DISPID_BACKCOLOR), helpstring("property BackColor")]
        HRESULT BackColor([out, retval] OLE_COLOR *pVal);
    [propput, id(DISPID_BACKCOLOR), helpstring("property BackColor")]
        HRESULT BackColor([in] OLE_COLOR newVal);
    [propget, id(dispidCaptionProp), helpstring("property
    CaptionProp")]
        HRESULT CaptionProp([in, optional] VARIANT varAlignment,
        [out, retval] BSTR *pVal);
    [propput, id(dispidCaptionProp), helpstring("property
    CaptionProp")]
        HRESULT CaptionProp([in, optional] VARIANT varAlignment,
        [in] BSTR newVal);
    [propget, id(dispidAlignment), helpstring("property Alignment")]
        HRESULT Alignment([out, retval] long *pVal);
    [propput, id(dispidAlignment), helpstring("property Alignment")]
        HRESULT Alignment([in] long newVal);
};

. . .
```

Of course, implementing the ReadyState property requires a member variable to store ReadyState's value. Add the m\_lReadyState member to the class declaration of the CATLControlWin class, as shown in Listing 32.2.

**Listing 32.2 ATLCONTROLWIN.H—Adding m\_lReadyState to the CATLControlWin Class Definition**

```
int iCharWidthArray[256];
int iCharacterSpacing, iCharacterHeight;

// for the ReadyState property
long m_lReadyState;
};
```

Don't forget to initialize the m\_lReadyState member variable in the constructor of the CATLControlWin class; if you don't, the asynchronous transfers might not progress as you'd expect!

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The last step is to implement the `get_ReadyState` function, which simply returns the current value of the `m_lReadyState` member variable, as shown in Listing 32.3.

### Listing 32.3 ATLCONTROLWIN.CPP—Implementing the `get_ReadyState` Function so that It Returns the `m_lReadyState` Member

```
STDMETHODIMP CATLControlWin::get_ReadyState(long * pVal)
{
    // set the return value to the value of the member variable
    *pVal = m_lReadyState;

    return S_OK;
}
```

### Supporting the *ReadyStateChange* Event

Now that you've added the `ReadyState` property, you still need to provide a notification hook to signal when `ReadyState` changes. You can do this with the `ReadyStateChange` event. Open the `ATLControl.idl` file, and add the `ReadyStateChange` function to the event interface, as shown in Listing 32.5.

### Listing 32.4 ATLCONTROLWIN.IDL—Adding the `ReadyStateChange` Event to the IDL File so that It Can Be Triggered

```
. . .

[
    uuid(C31D4C71-7AD7-11d0-BEF6-00400538977D),
    helpstring("ATLControlWin Event Interface")
]
dispinterface _DATLControlWin
{
    properties:
    methods:
        [id(1)] void Change([in, out]BSTR * bstrCaption,
            [in, out] long * lAlignment);
        [id(DISPID_READYSTATECHANGE)] void ReadyStateChange();
};

. . .
```

Remember that ATL doesn't automatically support events—you must manually rebuild the `CPATLControl.h` file that contains your connection point interface definitions. In Chapter 31, you added support for the `Change` event. To add support for `ReadyStateChange`, you must rebuild the file that defines the connection point interfaces so that the connection points include *both* events. To do this, follow these steps:

1. Compile the IDL file; you must do this first, because the event interface header file is built from the type library.
2. From the Project menu, select the Add to Project menu item, and then select the Components and Controls submenu item.
3. In the Components and Controls Gallery dialog box, double-click the Developer Studio Components



folder, and then double-click the ATL Proxy Generator icon. Click OK in the confirmation dialog that asks whether you want to insert the Proxy Generator.

4. When the ATL Proxy Generator dialog box appears, click the ... button to display the Open dialog box. Select the ATLControl.tlb file, and click Open.
5. Select the DATLControlWin entry in the Not Selected list box, and click the > button to move the entry to the Selectd list box. Ensure that the Proxy Type is set to Connection Point and click Insert.
6. A Save dialog box will appear with the file CPATLControl.h in the File name edit box. Click Save to continue. Click Yes to replace the existing CPATLControl.h file.
7. Click OK in the confirmation dialog box that indicates the operation was successful.
8. Close the ATL Proxy Generator and Components and Controls Gallery dialog boxes.

The Fire\_ReadyStateChange method has now been added to the CProxy\_DATLControlWin class.

Asynchronous properties are based on URLs, not on the type of data to be downloaded. The URL associated with each asynchronous property should be stored in a string property of the control. For our control, the property is called TextDataPath, and you need to add it to the control. From the ClassView tab in the Project Workspace window, select the IATLControlWin interface, click the right mouse button, and select the Add Property& menu item.

In the Add Property to Interface dialog box, set the Property Type to BSTR, the Property Name to TextDataPath, and leave the remainder of the settings at their default values (see Figure 32.2). Click OK to confirm the entry and close the dialog box.



**FIG. 32.2** Add the TextDataPath property to the control by using the ATL ClassWizard.

Because you added a new property, you should also add a new dispid constant, dispidTextDataPath, to the PROPDISPIDS enumeration in the ATLControl.idl file, and then update the TextDataPath function to use the constant value (see Listing 32.5).

#### **Listing 32.5 ATLCONTROL.IDL—Adding the dispidTextDataPath Enumeration to the IDL File**

```
. . .
typedef enum propdispid
{
    dispidAlignment = 2,
    dispidCaptionProp = 3,
    dispidTextDataPath = 4,
}PROPDISPIDS;

. . .
interface IATLControlWin : IDispatch
{
    [id(1), helpstring("method CaptionMethod")] HRESULT CaptionMethod(
        [in] BSTR bstrCaption, [in, optional] VARIANT varAlignment,
        [out, retval] long * lRetVal);
    [propget, id(dispidTextDataPath), helpstring("property
    0TextDataPath")]
        HRESULT TextDataPath([out, retval] BSTR *pVal);
    [propput, id(dispidTextDataPath), helpstring("property
    1TextDataPath")]
        HRESULT TextDataPath([in] BSTR newVal);
. . .
```

The TextDataPath property is used to store the URL of the data that the property represents. To complete your implementation of the property, add a member variable named m\_bstrTextDataPath to keep the URL available to

the class. This member is declared to be of type CComBSTR, which is a BSTR wrapper class provided with ATL; see the ATL documentation for more information. We've chosen to use CComBSTR to demonstrate the different implementation styles available to you with ATL. Instead, you could use an LPTSTR or BSTR without difficulty.

For this example, we're assuming our URL will always return a block of text, so you need to add the `m_bstrText` member variable, also of type CComBSTR, to store the data as it is downloaded. You'll see these two members used in "Transferring the Data" later in the chapter; Listing 32.6 shows their declarations.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Static and Dynamic Property Enumeration

Property enumeration restricts a property's values to a specific set of values. For example, a control that supports a property called TextAlignment might use values of 0, 1, and 2 for left-, center-, and right-justified text, or it could use enumerated values named LeftJustified, RightJustified, and CenterJustified. Another example is a property used to determine what languages a control supports. A language-based property might be a good candidate for both a static set, say English and German, and a dynamic set for all the languages available on a particular machine.

Property enumeration makes it easier for script writers to drive your control because they can write natural-sounding expressions like `theText.Justification = CenterJustified` instead of relying on meaningless integer values.

You can create property enumerations in two ways. The simplest way is to use *static enumeration*, where the enumerations are defined in the control's ODL or IDL file. As soon as the control is compiled, static enumerations stay fixed. The ATLControlWin version in Chapter 31 used this approach. The alternative, *dynamic enumeration*, is a little more complicated, but it enables you to change which enumerated values you support at runtime.

### Supporting Dynamic Property Enumeration

Dynamic property enumeration is based on the interface `IPerPropertyBrowsing`, which provides a standardized way for containers to get lists of available enumerated values. The ATL Object Wizard won't add this interface to your control class automatically, so you'll have to add it yourself. The first step is to add the `IPerPropertyBrowsingImpl` class to your control inheritance structure:

```
public IPerPropertyBrowsingImpl<CATLControlWin>
```

Next, add the `IPerPropertyBrowsing` interface to the COM interface map:

```
COM_INTERFACE_ENTRY_IMPL(IPerPropertyBrowsing)
```

The final step is to implement the actual `IPerPropertyBrowsing` interface functions. First, add the function prototypes to the class declaration, as shown in Listing 32.10.

#### Listing 32.10 ATLCONTROLWIN.H—Four Functions that Implement the `IPerPropertyBrowsing` Interface on Your Control

```

. . .
    STDMETHOD(MapPropertyToPage)(DISPID dispID, CLSID *pClsid);
    STDMETHOD(GetPredefinedStrings)(DISPID dispID, CALPOLESTR
    ð*pCaStringsOut,

        CADWORD *pCaCookiesOut);
    STDMETHOD(GetPredefinedValue)(DISPID dispID, DWORD dwCookie, VARIANT*
    ðpVarOut);
    STDMETHOD(GetDisplayString)(DISPID dispID, BSTR *pBstr);
. . .
  
```

### Getting the Right Property Page

`MapPropertyToPage` (see Listing 32.11) is used to link a property to a specific control or system-defined property

page. In this case, you return E\_NOTIMPL if the requested dispid matches that of the Alignment property. By returning E\_NOTIMPL, you are preventing the container application from displaying the property page associated with the property; instead, the container will use the property enumeration that you have implemented. The connection between the property and the property page is made in the property map macro in the class declaration. Remember that one of the parameters in the macro was the CLSID of the property page for the property.

---

**Listing 32.11 ATLCONTROLWIN.CPP—Using MapPropertyToPage to Handle Translation Between a Requested Property and the Appropriate Page**

---

```
STDMETHODIMP CATLControlWin::MapPropertyToPage(DISPID dispID, CLSID *pClsid)
{
    // if this is the dispid property
    if(dispID == dispidAlignment)
        // defer to the property enumeration and not the property page
        return E_NOTIMPL;
    else
        // defer to the base class implementation
        return IPerPropertyBrowsingImpl<CATLControlWin>::
            MapPropertyToPage(dispID, pClsid);
}
```

---

### Getting the Enumerations' String Values

GetPredefinedStrings is the first function to be called (see Listing 32.12). When this method is called, the dispid of the property that is currently being referenced will be passed in. This method is called for all properties that the control supports, so take care when adding code. If the function is called with a dispid that the control owns, the control is required to create an array of strings and *cookies*. A cookie is just a 32-bit value that has some meaning to the control implementation. In this case, the cookies that you supply are the enumeration constants that you added in Chapter 31: EALIGN\_LEFT, EALIGN\_RIGHT, and EALIGN\_CENTER. The strings are placed in a list from which the user of the control can select the appropriate value for the property.

---

**Tip:**

GetPredefinedStrings uses string literals defined in AlignmentEnumerations.h for its strings. In your controls, consider using string tables; not only does this isolate the effect of changing the strings, but it makes it easier to internationalize your controls.

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## Listing 32.12 ATLCControlWin.CPP—GetPredefinedStrings Returns the Strings Corresponding to the Available Enumerated Values

```

STDMETHODIMP CATLControlWin::GetPredefinedStrings(DISPID dispid,
    CALPOLESTR * lpcaStringsOut, CADWORD * lpcaCookiesOut)
{
    USES_CONVERSION;
    HRESULT hResult = S_FALSE;
    // we should have gotten two pointers if we didn't
    if((lpcaStringsOut == NULL) || (lpcaCookiesOut == NULL))
        // we are out of here
        return E_POINTER;
    // if this is the property that we are looking for
    if(dispid == dispidAlignment)
    {
        ULONG ulElems = 3;
        // allocate the memory for our string array
        lpcaStringsOut->pElems = (LPOLESTR *) ::CoTaskMemAlloc(
            sizeof(LPOLESTR) * ulElems);
        // if we couldn't allocate the memory
        if(lpcaStringsOut->pElems == NULL)
            // we're out of here
            return E_OUTOFMEMORY;
        // allocate the memory for our cookie array
        lpcaCookiesOut->pElems = (DWORD*) ::CoTaskMemAlloc(sizeof(DWORD*) *
            ulElems);
        // if we couldn't allocate the memory
        if (lpcaCookiesOut->pElems == NULL)
        {
            // free the string array
            ::CoTaskMemFree(lpcaStringsOut->pElems);
            // exit the function
            return E_OUTOFMEMORY;
        }
        // store the number of elements in each array
        lpcaStringsOut->cElems = ulElems;
        lpcaCookiesOut->cElems = ulElems;
        // allocate the strings
        lpcaStringsOut->pElems[0] = ATLA2WHELPER((LPWSTR)::CoTaskMemAlloc(
            (lstrlen(EALIGN_LEFT_TEXT) + 1) * 2), EALIGN_LEFT_TEXT,
            lstrlen(EALIGN_LEFT_TEXT) + 1);
        lpcaStringsOut->pElems[1] = ATLA2WHELPER((LPWSTR)::CoTaskMemAlloc(
            (lstrlen(EALIGN_CENTER_TEXT) + 1) * 2), EALIGN_CENTER_TEXT,
            lstrlen(EALIGN_CENTER_TEXT) + 1);
        lpcaStringsOut->pElems[2] = ATLA2WHELPER((LPWSTR)::CoTaskMemAlloc(
            (lstrlen(EALIGN_RIGHT_TEXT) + 1) * 2), EALIGN_RIGHT_TEXT,
            lstrlen(EALIGN_RIGHT_TEXT) + 1);
        // assign the cookie value
        lpcaCookiesOut->pElems[0] = EALIGN_LEFT;
        lpcaCookiesOut->pElems[1] = EALIGN_CENTER;
        lpcaCookiesOut->pElems[2] = EALIGN_RIGHT;
        hResult = S_OK;
    }
}
    
```

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

```

    }
    return hResult;
}

```

---

## Getting an Enumeration Value

GetPredefinedValue, shown in Listing 32.13, is called when the container's property browser needs the real enumeration value associated with a particular dispid and cookie. The value returned is the actual value that is stored in the property, not the string that was used to represent it. For our control, the returned value will always be a VT\_I4, but if, for example, we were storing a string instead, we'd return a string variant.

### Listing 32.13 ATLCONTROLWIN.CPP—Using GetPredefinedValue to Map the Control-Defined Cookie into a Variant

---

```

STDMETHODIMP CATLControlWin::GetPredefinedValue(DISPID dispid, DWORD dwCookie,
    VARIANT* lpvarOut)
{
    BOOL bResult = FALSE;
    // which property is it
    switch(dispid)
    {
    case dispidAlignment:
        // clear the variant
        ::VariantInit(lpvarOut);
        // set the type to a long
        lpvarOut->vt = VT_I4;
        // set the value to the value that was stored with the string
        lpvarOut->lVal = dwCookie;
        // set the return value
        bResult = TRUE;
    break;
    }
    return bResult;
}

```

---

## Turning an Enumeration into a String

After the container calls GetPredefinedValue, it can set the property with the correct value—but it still needs to map that value to a human-readable string. GetDisplayString does just that: It returns the string that is associated with the current property setting.

### Listing 32.14 ATLCONTROLWIN.CPP—Tying an Enumeration Back to a Text String with GetDisplayString

---

```

STDMETHODIMP CATLControlWin::GetDisplayString(DISPID dispid, BSTR* lpbstr)
{
    USES_CONVERSION;
    HRESULT hResult = S_FALSE;
    // which property is it
    switch(dispid)
    {
    case dispidAlignment:
    {
        switch(m_lAlignment)
        {
        case EALIGN_LEFT:
            *lpbstr = ::SysAllocString(T2OLE(EALIGN_LEFT_TEXT));
            break;
        case EALIGN_CENTER:
            *lpbstr = ::SysAllocString(T2OLE(EALIGN_CENTER_TEXT));
            break;

```

```
        case EALIGN_RIGHT:
            *lpbstr = ::SysAllocString(T2OLE(EALIGN_RIGHT_TEXT));
            break;
        }
        // set the return value
        hResult = S_OK;
    }
    break;
}
return hResult;
}
```

---

#### Note:

The property browser already gets a string for every value when it calls `GetPredefinedStrings`, so `GetDisplayString` might seem a little redundant. You should have `GetDisplayString` because it can be implemented without implementing the other methods. Property types that do not use the standard property selection mechanism (for example, fonts) can use `GetDisplayString` to fill out a list in a dialog.

---

## Optimizing Control Drawing

*Optimized drawing* enables you to create drawing objects, such as pens or brushes, once. Rather than removing them when you are finished drawing, you can cache them as control member variables and reuse them each time your control draws itself. The benefit is that you create drawing resources once for the lifetime of your control, rather than every time it draws.

If your control doesn't do a lot of custom drawing, optimized drawing might not boost its performance. There's also a size-versus-speed tradeoff between standard and optimized drawing: standard drawing doesn't use member variables for the drawing objects that are created and used, so it saves space at the expense of having to reallocate the drawing objects each time the control is drawn. Optimized drawing trades that space for the increased speed resulting from reusing graphic resources in the redraw loop.

There's an additional drawback to optimized drawing: The container that your control ends up in might not support it. Your controls must continue to support standard drawing functionality, switching to optimized only if it is available.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

#### Note:

ATL's optimized drawing capabilities are only a subset of those defined in the OC 96 specification. The OC 96 specification further breaks optimized drawing into *aspects*. For more information on aspect drawing, please see the OC 96 Specification that ships with the ActiveX SDK.

In Chapter 31, you learned how to implement standard drawing so that your control could draw itself. Supporting optimized drawing is fairly simple, as you'll see.

First, you need to check whether or not the container supports optimized drawing. The `bOptimize` member of the `ATL_DRAWINFO` structure passed to your draw method will be `FALSE` if the container can't handle optimized drawing. In that case, you have to clean up all of your allocated resources and restore any original values. If the container *does* support optimized drawing, you ignore the cleanup and reuse the allocated resources the next time around. Listing 32.15 shows the modified `OnDraw` function.

#### Listing 32.15 ATLCONTROLWIN.CPP—Optimizing Drawing by Caching Drawing Resources—if the Container Supports It

```

HRESULT CATLControlWin::OnDraw(ATL_DRAWINFO & di)
{
    . . .
    // The container does not support optimized drawing.
    if(!di.bOptimize)
    {
        // select the old brush back
        ::SelectObject(di.hdcDraw, hOldBrush);
        // destroy the brush we created
        ::DeleteObject(hBrush);
        // clear the brush handles
        hBrush = hOldBrush = NULL;
    }
    return S_OK;
}
  
```

To clean up the cached variables, you must add a message handler, `OnDestroy`, for the Windows message `WM_DESTROY`. Also add an `OnDestroy` function prototype to the class declaration, as shown in Listing 32.16. `OnDestroy` is used to clean up the drawing resources if any are still allocated when the control is destroyed. This should happen *only* if the container supports optimized drawing; if not, the resources should be freed normally when drawing is done.

#### Listing 32.16 ATLCONTROLWIN.H—Adding WM\_DESTROY Support to Enable Cleanup

```

BEGIN_MSG_MAP(CATLControlWin)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_GETDLGCODE, OnGetDlgCode)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
    MESSAGE_HANDLER(WM_KILLFOCUS, OnKillFocus)
    MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
END_MSG_MAP()

. . .
LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL & bHandled);
. . .
  
```

The last step is to add the `OnDestroy` implementation, which will clean up the resources if they are still allocated (see



**Listing 32.17 ATLCONTROLWIN.CPP—Cleanup Performed by OnDestroy**

---

```

LRESULT CATLControlWin::OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL &
bHandled)
{
    // if there is an old brush
    if(hOldBrush)
    {
        // get the DC
        HDC hDC = this->GetDC();
        // select the old brush back
        ::SelectObject(hDC, hOldBrush);
        // release the DC
        this->ReleaseDC(hDC);
    }
    // if we created a brush
    if(hBrush)
        // destroy the brush we created
        ::DeleteObject(hBrush);
    return TRUE;
}

```

---

## Adding Clipboard and Drag-and-Drop Support

The basic OLE Clipboard and Drag-and-Drop interfaces are only partially implemented by the default ATL control implementation. As with the IPerPropertyBrowsing interface, you must provide the remaining required interfaces yourself. Supporting the Clipboard and Drag-and-Drop in your control adds polish and ease-of-use for the end user with relatively little effort on your part.

### Talking to the Clipboard

Clipboard support in ActiveX controls comes from the IDataObject and IEnumFORMATETC interfaces. IDataObject provides an interface for the Clipboard to request data from your control, and IEnumFORMATETC is the interface used by other applications to determine what data types your control's IDataObject interface supports. In a basic ATL control project, only the IDataObject interface is supported. The IEnumFORMATETC interface must be added.

Before adding the specific interfaces required by ActiveX to enable Clipboard transfers, you must first decide which keystroke combinations will be used to initiate the cut, copy, or paste operations. You should either use Ctrl+X, Ctrl+C, and Ctrl+V or Shift+Delete, Ctrl+Insert, and Shift+Insert for cut, copy, and paste, respectively.

**Watching for the Editing Keys.** To trap these keystrokes in your control, you'll need a message handler for the WM\_KEYDOWN message in the form of a method called OnKeyDown (see Listing 32.18).

**Listing 32.18 ATLCONTROLWIN.H—Adding WM\_KEYDOWN to the Message Map and then Adding an OnKeyDown Handler**

---

```

. . .
BEGIN_MSG_MAP(CATLControlWin)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_GETDLGCODE, OnGetDlgCode)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
    MESSAGE_HANDLER(WM_KILLFOCUS, OnKillFocus)
    MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
    MESSAGE_HANDLER(WM_KEYDOWN, OnKeyDown)
END_MSG_MAP( )
. . .
LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL & bHandled);
LRESULT OnKeyDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL & bHandled);
. . .

```

---

The OnKeyDown method, shown in Listing 32.19, looks for the predefined editing keystroke combinations listed in the

preceding paragraph; if the method finds one, it calls the inherited data transfer functions to complete the requested Clipboard operation. If the user requested a Cut operation, OnKeyDown fires the Change and ViewChange events to redisplay the control after clearing its data.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 [EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)
[Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

You might have noticed that the version of OnKeyDown shown in Listing 32.19 didn't support pasting from the Clipboard. The code in Listing 32.31 shows the changes needed to support pasting with either Ctrl+V or Shift+Insert. In either case, OnKeyDown calls GetDataFromClipboard to get the data and forces the control to update by firing a message with FireViewChange.

### Listing 32.31 ATLCONTROLWIN.CPP—OnKeyDown Implementation

```

...
switch(nChar)
{
// PASTE
case 0x56: // 'V'
case 0x76: // 'v'
    // if the control key is being held down
    if(sControl & 0x8000)
    {
        // get any text from the clipboard
        this->GetDataFromClipboard();
        // force the control to repaint itself
        this->FireViewChange();
    }
    // we don't need to pass this key to the base implementation
    bHandled = TRUE;
}
break;
// COPY or PASTE
case 0x43: // 'C'
case 0x63: // 'c'
case VK_INSERT:
    // if the control key is being held down
    if(sControl & 0x8000)
    {
        // copy the data to the clipboard
        this->CopyDataToClipboard();
        // we don't need to pass this key to the base implementation
        bHandled = TRUE;
    }
    // if the shift key is being held down it is a PASTE
    else if(sShift & 0x8000 && nChar == VK_INSERT)
    {
        // get any text from the clipboard
        this->GetDataFromClipboard();
        // force the control to repaint itself
        this->FireViewChange();
    }
    // we don't need to pass this key to the base implementation
    bHandled = TRUE;
}
break;
...

```

## Supporting Drag-and-Drop

The fundamentals of supporting Drag-and-Drop are very similar to those needed for Clipboard support.

Drag-and-Drop relies on the same set of interfaces for the actual data transfer, but it adds two new interfaces: IDropSource and IDropTarget. **Drag sources**—controls that create data that can be dropped onto another application—need to be inherited from IDropSource. The corresponding IDropTarget is for controls that can act as **drop targets**; targets accept data that has been dropped from another application or control.

Because Drag-and-Drop is essentially a Clipboard transfer—with fewer steps involved—Drag-and-Drop uses the same built-in data formats as Clipboard transfers, and the overall implementation is quite similar.

---

**Tip:**

If it doesn't make sense for your control to be only a drag target or drag source but not both, don't worry. The two are independent, so you can implement either or both.

---

**Enabling a Control as a Drag-and-Drop Source.** The first part of your implementation is to enable the control as a drag source. To act as a drag source, the control must implement the IDropSource interface, in addition to the IDataObject and IEnumFORMATETC interfaces that you implemented in the “Supporting the Clipboard” section.

ATLControlWin inherits from IDropSource in the same manner as the other COM interfaces that you've added previously. Because the interface consists of only two methods, the implementation is easy. QueryContinueDrag is called by the system event handler so that you can tell it whether or not to register the drop event and send it to the target. GiveFeedback is called during the drag, so you can change the displayed cursor or otherwise indicate to the user that a drag is in progress. The necessary changes to ATLControlWin.h are shown in Listing 32.32.

---

**Listing 32.32 ATLCONTROLWIN.H—IDropSource Interface Added to the CATLControlWin Class Declaration**

---

```
. . .
    public IEnumFORMATETC,
    public IDropSource
{
public:
    CATLControlWin()
. . .
    COM_INTERFACE_ENTRY( IEnumFORMATETC )
    COM_INTERFACE_ENTRY( IDropSource )
END_COM_MAP()
. . .
// IEnumFORMATETC
. . .
// IDropSource
    STDMETHOD(QueryContinueDrag)(BOOL fEscapePressed, DWORD dwKeyState);
    STDMETHOD(GiveFeedback)(DWORD dwEffect);
. . .
```

---

Before you can actually initiate a drag, you need to know when the user presses the left mouse button in your control. The easiest way to do this is to catch the WM\_LBUTTONDOWN message, so you'll need a message handler for it. Add WM\_LBUTTONDOWN message handler called OnLButtonDown to the message map, as shown in Listing 32.33.

---

**Listing 32.33 ATLCONTROLWIN.H—WM\_LBUTTONDOWN and OnLButtonDown Message Handler Added to the Class Declaration of the Control**

---

```
. . .
BEGIN_MSG_MAP( CATLControlWin )
. . .
    MESSAGE_HANDLER( WM_LBUTTONDOWN, OnLButtonDown )
END_MSG_MAP()
. . .
    LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL &
bHandled);
    LRESULT OnKeyDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL &
bHandled);
```

```
LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL &
bHandled);
. . .
```

---

The OnLButtonDown implementation, as shown in Listing 32.34, is similar to the CopyDataToClipboard method shown in Listing 32.25. OnLButtonDown prepares the data for its transfer and calls the Win32 DoDragDrop function to start the drag. Note that we always pass DROPEFFECT\_COPY as the drop modifier for DoDragDrop; you could check for modifier keys and pass a different drop modifier to enable users to modify what happens on a drag.

#### **Listing 32.34 ATLCONTROLWIN.CPP—OnLButtonDown Implementation**

---

```
LRESULT CATLControlWin::OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam,
    BOOL & bHandled)
{
    //      Un-comment these parameters if you need them
    //      UINT nFlags = wParam;
    //      short sHor = (short) LOWORD(lParam);
    //      short sVer = (short) HIWORD(lParam);

    // call the common data preparation function
    this->PrepareDataForTransfer();
    DWORD dwDropEffect = DROPEFFECT_NONE;

    // start the Drag and Drop operation
    ::DoDragDrop(reinterpret_cast<IDataObject*>
        (static_cast<IDataObjectImpl<CATLControlWin*>>(this)),
        (IDropSource *) this, DROPEFFECT_COPY, &dwDropEffect);
    return TRUE;
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Listing 32.40 ATLCONTROLWIN.CPP—DragEnter: Allowing You to Decide Whether to Accept or Reject a Drag

```

STDMETHODIMP CATLControlWin::DragEnter(LPDATAOBJECT pDataObject, DWORD dwKeyState,
    POINTL pt, LPDWORD pdwEffect)
{
    // if the left mouse button is being held down
    if(dwKeyState & MK_LBUTTON)
    {
        IEnumFORMATETC * ipenumFormatetc;
        BOOL bFound = FALSE;
        // get a FORMATETC enumerator
        if(pDataObject->EnumFormatEtc(DATADIR_GET, &ipenumFormatetc) ==
            S_OK)
        {
            // reset the enumerator just to be safe
            ipenumFormatetc->Reset();
            FORMATETC etc;
            // while there are formats to enumerate
            while(ipenumFormatetc->Next(1, &etc, NULL) == S_OK && !bFound)
            {
                // is this a format that we are looking for?
                if(etc.cfFormat == CF_TEXT && etc.tymed & TYMED_HGLOBAL)
                    bFound = TRUE;
            }
            // release the enumerator
            ipenumFormatetc->Release();
        }
        // is there a text format available
        if(bFound)
            *pdwEffect = DROPEFFECT_COPY;
        // everything else we can't deal with
        else
            *pdwEffect = DROPEFFECT_NONE;
    }
    else
        // not the left mouse
        *pdwEffect = DROPEFFECT_NONE;
    // return success
    return S_OK;
}
    
```

DragOver, shown in Listing 32.41, is used to test the current state of the drag operation while it is over the control. This implementation is very basic. One could, however, change the method to restrict the drag operation. You might check the POINTL structure that was passed in and compare it to various locations of the control. As another example, a grid or “spreadsheet” control might enable only text data to be dropped on the headings but both text and numeric data while over the columns.

### Listing 32.41 ATLCONTROLWIN.CPP— Constraining the Drag, Based on the Mouse Coordinates

```

STDMETHODIMP CATLControlWin::DragOver(DWORD dwKeyState, POINTL pt,
    LPDWORD pdwEffect)
{
    // if the left mouse button is being held down
    if(dwKeyState & MK_LBUTTON)
    
```

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

```

        // copy
        *pdwEffect = DROPEFFECT_COPY;
    else
        // not the left mouse
        *pdwEffect = DROPEFFECT_NONE;
    // return success
    return S_OK;
}

```

---

DragLeave is used to clean up any state information that might have been created when DragEnter was invoked. If you don't allocate anything during DragLeave, just return E\_NOTIMPL, as we do for ATLControlWin.

Drop (see Listing 32.42) is the last function that you need to implement. It actually copies the data from the source IDataObject to the control by using the GetDataFromTransfer method that we implemented for Clipboard support.

---

#### Listing 32.42 ATLCONTROLWIN.CPP—IDropTarget Interface Implementation

---

```

STDMETHODIMP CATLControlWin::Drop(LPDATAOBJECT pDataObject, DWORD dwKeyState,
    POINTL pt, LPDWORD pdwEffect)
{
    // transfer the data to the control
    this->GetDataFromTransfer(pDataObject);
    // return success
    return S_OK;
}

```

---

That's it! If all you want is basic Drag-and-Drop support, you're done. You can also support custom formats, however, with a few more lines of code.

### Handling Custom Clipboard and Drag-and-Drop Formats

In addition to the standard Windows formats, applications and controls can define their own *custom data formats* for data exchange. For example, Intergraph's Imagineer computer-aided design software defines custom formats for 2-D and 3-D drawings. When transferring data via Drag and Drop or the Clipboard, Imagineer supplies data in metafile, DIB, and its own custom format. Applications that understand the Imagineer formats can use them; applications that don't can use the metafile or DIB version of the data instead.

Support for custom data formats is independent of the mechanism used to initiate the data transfer. Because we isolated the methods used to prepare data for transfer from those that actually do the transferring, we need make only one set of changes to accommodate custom Clipboard and Drag-and-Drop operations.

In preceding sections of this chapter, you implemented support for transferring ATLControlWin's caption as plain text. Next you examine what would be required to transfer the custom Alignment property along with the Caption.

**Registering a Custom Format.** Windows already provides predefined IDs for the common CF\_XXX formats that it provides. To make your custom formats visible, you'll have to register them with Windows. This requires you to keep track of the ID number of the registered custom format; add a member variable named m\_uiCustomFormat for this purpose. In addition, you also need an extra FORMATETC/STGMEDIUM pair so that you can store your custom data *and* text data simultaneously. Listing 32.43 shows the required change.

---

#### Listing 32.43 ATLCONTROLWIN.H—Custom Data Format Member Variables

---

```

...
private:
    FORMATETC sTextFormatEtc;
    STGMEDIUM sTextStgMedium;
    // custom format storage variables
    UINT m_uiCustomFormat;
    FORMATETC sCustomFormatEtc;
    STGMEDIUM sCustomStgMedium;
};

```

---

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

The final requirement is actually returning data to a caller. IEnumFORMATETC::GetData has to be modified to return data from our text STGMEDIUM when text is requested and from our custom format's STGMEDIUM when the custom format is requested. The new version is in Listing 32.48. Note that you can still use the CopyStgMedium function; the only difference is which of CATLControlWin's cached STGMEDIUMs you call it with.

#### Listing 32.48 ATLCONTROLWIN.CPP—IEnumFORMATETC::GetData Update

```

STDMETHODIMP CATLControlWin::GetData(LPFORMATETC lpFormatEtc, LPSTGMEDIUM
lpStgMedium)
{
    // if this is a format that we can deal with
    if(lpFormatEtc->cfFormat == CF_TEXT && lpFormatEtc->tymed &
    TYMED_HGLOBAL)
    {
        // get a copy of the current stgmedium
        this->CopyStgMedium(lpStgMedium, &sTextStgMedium, CF_TEXT);
        return S_OK;
    }
    else if(m_uiCustomFormat && lpFormatEtc->cfFormat == m_uiCustomFormat
    && lpFormatEtc->tymed & TYMED_HGLOBAL)
    {
        // get a copy of the current stgmedium
        this->CopyStgMedium(lpStgMedium, &sCustomStgMedium,
    m_uiCustomFormat);
        return S_OK;
    }
    else
        return IDataObjectImpl<CATLControlWin>::GetData(lpFormatEtc,
    lpStgMedium);
}
    
```

That's all it takes to support custom formats. By taking a "black box" approach and separating the data transfer routines from the data preparation routines, you can easily add support for other formats without extensive code changes.

## ATL Support for Other Advanced ActiveX Features

ATL enables you to take advantage of many of the available OC 96 and ActiveX features. In some cases, support for these features is automatic.

### Dual-Interface Controls

ActiveX controls can communicate with other applications in two ways: using COM interfaces, like IDataObject, or by using interfaces derived from IDispatch. ActiveX automation, discussed in detail in Chapter 29, "Creating ActiveX Controls," depends on IDispatch. Without ATL, you'd have to manually add support for IDispatch calls, which is not a light undertaking. By default, ATL control implementations support both kinds of interface, so you do not need to do any additional work. If for some reason you want to generate a control with only one type of interface, you can do so in the Attributes tab of the ATL Object Wizard Properties

Brief Full

- Advanced Search
- Search Tips

BROWSE  
BY TOPIC

dialog box.

## Windowless Activation

ActiveX controls can request that no new window be generated when they're in-place activated. In this case, the container can receive window messages for the control and dispatch them without requiring the control to have its own window. The `IOleInPlaceObjectWindowless` interface, implemented in the container, provides this capability. If the container doesn't support windowless activation, the control must be able to create a window for itself. Windowless activation is a request, and there's a guarantee that all containers will grant it.

ATL supports windowless controls out-of-the-box. When generating a new control, use the Miscellaneous tab of the ATL Object Wizard Properties dialog box to clear the `Windowed Only` check box if you want the control to support windowless activation. The `CComControl` class contains a member variable, `m_bWindowOnly`, which, if set to `TRUE`, instructs the control to use windowless activation if the container supports it. If set to `FALSE`, the control will always create a window handle and use it.

## Unclipped Device Context

Unclipped device context is an MFC-specific optimization and is not yet implemented in ATL.

## Flicker-Free Activation

*Flicker-free* activation requires that the container support the `IOleInPlaceSiteEx` interface. ATL automatically attempts to find this interface and will use it if the container supports it. Flicker-free Activation requires no implementation on the part of the developer.

## Mouse Pointer Notifications when Inactive

If you need to, you can request notification when the mouse pointer moves over your control while it's inactive. These notifications are provided through the `IPointerInactive` COM interface. To support this interface in your control, you must add the ATL class `IPointerInactiveImpl` to your class declaration and override the ATL implementations of the `GetActivationPolicy`, `OnInactiveMouseMove`, and `OnInactiveSetCursor` methods.

## Optimized Drawing Code

Optimized drawing is fairly simple to implement, as you saw in the "Enabling Optimized Drawing" section, earlier in this chapter. A parameter of the `OnDraw` method indicates whether or not the control is able to use optimized drawing. In addition, ATL supports *aspect*-optimized drawing, which is beyond the scope of this book. If you want to implement this feature, please see the OC 96 specification included in the ActiveX SDK.

## Summing Up

This chapter focused on expanding the basic control implementation that you created in Chapter 31 by adding support for some advanced ActiveX features. As the user and developer market for ActiveX controls becomes more discerning, these features will become increasingly important in separating successful controls from unsuccessful ones. Fortunately, ATL provides solid support for these features and minimizes the amount of effort required for you to implement them.

The next three chapters explore using MFC for some traditional, but necessary, tasks: managing linked lists, performing recursive operations, and generating new classes from scratch.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Part VI

# MFC and Computer Science

## Chapter 33 Programming Linked Lists

- About John Conway's Game of Life
- How to program a linked list
- How to create a linked-list class
- How to use linked lists to speed up the Life application
- How to program the Game of Life for Windows 95

Now that you know how to whip your MFC programs into shape, you'll spend the last few chapters of this book learning some new techniques that'll help you write more professional and effective applications. **Linked lists**, for example—which are the subject of this chapter—provide an excellent structure for storing certain types of data. Linked lists are so important and are used so often in professional programs that MFC includes classes for handling various types of linked lists. These classes are CObList, CStringList, and CPtrList.

### The Story of Life

Over 30 years ago, a fine English fellow by the name of John Conway invented simple rules for a system that simulated the lives of special, one-celled animals. Although the rules of the simulation were simple, the results were fascinating. Before long, every computer scientist worth his or her degree had written a version of Conway's Game of Life and had spent hours trying different combinations of cells to see what patterns might emerge.

Today, people are still fascinated by Conway's computer simulation. Many computer science books at least mention Life, and each year thousands of computer science students write versions of Life as part of their programming curriculum. The simplest implementations result in programs that accurately portray the simulation but run too slowly to be practical. Other implementations blaze across the screen in vivid colors and kaleidoscopic patterns, hypnotizing any viewer who happens to glance in its direction.

What do linked lists and MFC have to do with a simulation of one-celled

BROWSE  
BY TOPIC

creatures? That's a question whose answer you'll know by the time you get to the end of this chapter. But before you get started with this chapter's programming, take a little time to read the next section, where you'll learn about Life and how the program works.

## The Rules of Life

The Life simulation is played on a grid of any size. In the original rules, the grid is unbounded, but you can limit the grid to the screen. You might want to think of the screen display as a sort of petri dish holding a culture of microscopic cells. Cells are placed randomly on the grid, and the simulation is started. The cells then run through their life cycles a given number of generations, living and dying according to the rules set forth by Conway.

The rules are simple and elegant: In any given generation, any live cell with less than two neighbors dies of loneliness. Any live cell with more than three neighbors dies of crowding. Any dead cell with exactly three neighbors comes to life. And, finally, any live cell with two or three neighbors lives, unchanged, to the next generation.

## Life Implementation

As you might imagine, a large grid could contain hundreds if not thousands of cells living and dying in every generation. The computer must work furiously, calculating the number of neighbors for each cell in the grid and then creating or killing cells based on these counts. And keep in mind that counting the neighbors for a single cell requires checking each adjacent cell—as many as eight.

Suppose you implemented the grid as a two-dimensional array of integers, like this:

```
int map[32][32];
```

Each element of the map can be one of two values: 0 if the cell is dead, and 1 if the cell is alive. The logical way to process this grid is to check each element of the array, counting its neighbors and marking it as alive or dead.

In the example 32×32 array, 1,024 cells must be processed every generation. Each cell processed must check the status of as many as eight adjacent cells to count its neighbors. That's over 8,000 operations for the entire grid. Worse, this processing must be performed for every generation of the simulation. A single run of the simulation might have as many as 10,000 generations!

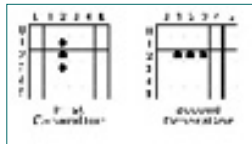
All of this calculating wouldn't be a problem if you planned to let the simulation run all night. However, to make the simulation interesting, you must update the screen as quickly as possible, ideally several times a second. Obviously, this creates a problem in the speed department.

## Creating and Killing Cells

Speed is not the only problem. You also must consider the effects of

prematurely creating or killing cells. It's not enough to scan through the grid, creating and killing cells as you go, because the cells that you create or kill might affect cells that you have not yet processed. Suppose cell X in a grid has only two neighbors. Now assume that a cell next to X dies as you process the grid. Although this cell died, cell X should still remain alive for this generation because it had two neighbors; it won't be lonely until the next generation. When you finally process cell X, however, the counting function recognizes cell X as having only one neighbor. As a result, cell X dies prematurely.

Confused? Look at Figure 33.1. Three cells are in the first-generation grid, which is on the left. In this generation, the uppermost cell must die because it has only one neighbor. The middle cell must remain alive until the next generation, because it has two neighbors. The bottom cell must die because, like the top cell, it has only one neighbor. The empty cells to the left and right of the center cell must be brought to life because both have exactly three neighbors. After processing the grid, you should have the second-generation grid, which is on the right.



**FIG. 33.1** Applying the rules of Life to three cells yields the results shown in the right hand grid.

However, if you start at the top and process the grid by creating and killing cells as you go, you'll get incorrect results. First, you kill the top cell, because it has only one neighbor. Then when you get to empty cell 1,2, even though it should have come to life, you determine that it has only two neighbors and leave it alone. When you get to cell 2,2, you think it has only one neighbor and kill it, even though this cell should have survived to the next generation. After processing the entire grid, you don't have the correct second-generation result. Instead, you have an empty grid!

In short, in each generation, you must determine which cells will live and die, without changing the grid. Then when you are finished, you must simultaneously create and kill the appropriate cells. This requires tricky algorithms, especially when you consider that all these calculations must be performed at a speed that allows fast screen updates. Sound like fun? Let's give it a shot.

## Dealing with the Speed Problem

What can you do to speed things up? First, add another map array to keep a running count of each cell's neighbors. When the simulation starts, the program will do a full update of the neighbor count. From then on, rather than recalculating the entire grid in each generation, the program changes neighbor counts for only those cells adjacent to cells that have just been created or killed. This method cuts processing time significantly: In a given generation, the program must change the neighbor counts of only a small number of cells instead of the entire grid.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Then, although the original map grid will record the status of each cell, add two lists of cells—one for cells about to be created and one for cells about to die. These are the only cells that affect the map, so there's no reason to check the entire grid every generation.

But what type of data structure enables you to build lists of items—lists that can grow or shrink dynamically? You've probably already guessed that the answer is a linked list.

## Linked Lists

Linked lists are an incredibly useful data structure for organizing information in memory. One of their big advantages is that you can add, remove, and move items in the list just by manipulating a couple of pointers. And because the contents of a linked list are determined dynamically as the program runs, you don't need to decide on a size for the list ahead of time, as you would with a data structure like an array. This enables you to utilize memory to its fullest. In the sections that follow, you'll learn the basic techniques used for creating and managing linked lists. This background will help you to better understand the linked-list classes provided by MFC.

### Creating a Linked List

To create a linked list, you first must decide what information makes up the items, or nodes, that will be stored in the list. In the simulation program, you must store enough data to identify a cell. All the information that you need to identify a cell are its X and Y coordinates in the grid, so a node could be the structure shown in Listing 33.1.

#### Listing 33.1 lst33\_01.cpp—A Structure for a Node

```
struct Node
{
    int x, y;
};
```

When a cell is born or dies, you can create a node for the cell like this:

```
Node *node = new Node;
node->x = x_ccord;
```



```
node->y = y_coord;
```

This code creates a new Node structure on the heap and sets its X and Y members to the coordinates of a cell. But what good is it to have a bunch of these nodes sitting around in memory? You must link them into a list. To do this, you must add to your structure a pointer to a node. You can then use this pointer to point to the next node in the list. The new Node structure, then, looks like Listing 33.2.

### Listing 33.2 lst33\_02.cpp—A Node for a Linked List

---

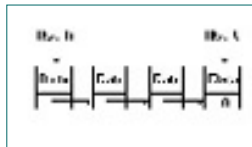
```
struct Node
{
    int x, y;
    Node *next;
};
```

---

In addition to the data structure for a node, you also need a pointer to the first node of the list (a **head pointer**) and a pointer to the end of the list (a **tail pointer**). Having a pointer to the head of the list is most important. Without it, you can't find the list in memory. A pointer to the tail is a convenience. You can use it to quickly add new nodes to the end of the list, without having to scan the list from the first node. The head and tail pointers look like this:

```
Node *list_h, *list_t;
```

Figure 33.2 illustrates how a linked list looks in memory. The list\_h pointer points to the first node in the list. Each node has a pointer that leads to the next node in the list. The next pointer in the last node is left NULL, which indicates the end of the list. Finally, the list\_t pointer points to the last node in the list.



**FIG. 33.2** This is what a linked list looks like in memory.

Listing 33.3 is the source code for a simple list-handling program. Note that this program is not meant to run under Windows and is presented here only to demonstrate the basic techniques of programming a linked list.

### Listing 33.3 lst33\_03.cpp—A Simple Linked List Demonstration

---

```
struct Node
{
    int x, y;
    Node *next;
};

Node *node = NULL,
```



```

        *list_h = NULL,
        *list_t = NULL;

void main(void)
{
    for (int i = 0; i < 10; ++i)
    {
        node = new Node;
        node->x = i;
        node->y = i * 10;
        if (!list_h)
            list_h = node;
        else
            list_t->next = node;
        list_t = node;
        list_t->next = NULL;
    }

    while (list_h)
    {
        node = list_h;
        list_h = list_h->next;
        cout << node->x << ',' << node->y << '\n';
        delete node;
    }
}

```

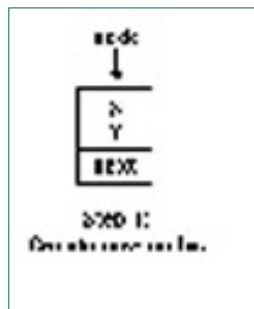
---

Study Listing 33.3 carefully, so you're sure you understand how to create and manage a linked list. This knowledge will help you better understand how to take advantage of MFC's list classes. In Listing 33.3, the Node structure is the type of item stored in the list.

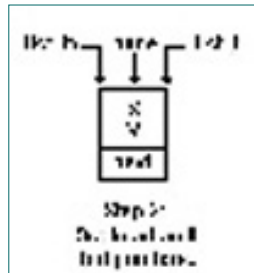
This structure contains two data elements, as well as a pointer to a Node. This pointer, next, is used to point to the next node in the list.

The program begins with a for loop, in which ten nodes are created and linked. In the loop, the new operator creates a new node on the heap, after which the node's data elements are set to the values of i and i\*10. (These values hold no particular significance.) After creating the node, the program checks whether list\_h is NULL. If it is, the program has a new list, so it sets list\_h to point to node. Then list\_t is set to point to the same node (if the list has only one item, the head and tail of the list are the same), and list\_t's next pointer is set to NULL, indicating there are no other items in the list.

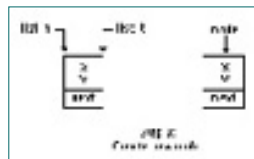
Getting back to the if statement, if list\_h isn't NULL, there's already at least one node in the list. In this case, list\_h shouldn't be changed. Rather, the new node must be added to the end of the list. This is where list\_t comes in handy. Rather than having to scan through the whole list, looking for a NULL next, the program can use list\_t to tack the new node onto the end of the list. It does this by setting list\_t's next pointer to point to the new node and then changing list\_t to point to the new last node. Figures 33.3 through 33.6 illustrate this process.



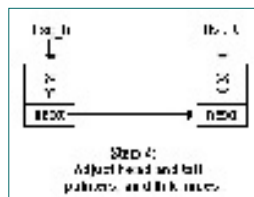
**FIG. 33.3** Step 1 of creating a linked list.



**FIG. 33.4** Step 2 of creating a linked list.



**FIG. 33.5** Step 3 of creating a linked list.



**FIG. 33.6** Step 4 of creating a linked list.

After the program creates the linked list, a while loop scans the list, printing each node's contents before deleting the node. Notice how the temporary node pointer keeps track of the current node. By setting node to list\_h and then setting list\_h to point to the next item in the list, you effectively “pop off” the first node. Without saving the pointer in node, you could not access this node. The program's output follows:

```
0,0
1,10
2,20
3,30
4,40
5,50
6,60
7,70
8,80
9,90
```

## An Object-Oriented List

If you've an idea that a linked list might be the perfect candidate for a class, you could be correct, depending on how you plan to use the list. Creating a linked list class to handle only a single list in a small program such as Listing 33.3 is overkill (or is it?). However, if you plan to use many different lists in a program—that is, the class won't be a single-instance class—it might be worthwhile to create a linked list class.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

For the sake of discussion, you'll now convert Listing 33.1 into an object-oriented program. Listing 33.4 is the header for the resultant List class.

#### Listing 33.4 lst33\_04.cpp—The Header File for the List Class

```
#ifndef _LIST_H
#define _LIST_H

class List
{
    struct Node
    {
        int x, y;
        Node *next;
    };

    Node *node, *list_h, *list_t;

public:
    List(void);
    ~List(void);
    void MakeNewNode(int n1, int n2);
    void DisplayList(void);
};
#endif
```

As you can see, all of the list-handling operations have been taken out of the main program and placed into the List class. The data that defines the list—the pointers and the node declaration—are placed inside the class also. The main program no longer has to know how a linked list works. It only has to draw on the capabilities of the class. First, look at the class's constructor, as shown in Listing 33.5.

#### Listing 33.5 lst33\_05.cpp—The List Class's Constructor

```
List::List(void)
{
    list_h = list_t = NULL;
}
```

This function initializes a new list by setting its pointers to NULL. This creates an empty list—which isn’t particularly useful. The class needs a way to add nodes to the list, as shown in Listing 33.6.

---

**Listing 33.6 lst33\_06.cpp—The MakeNewNode() Member Function**

---

```
void List::MakeNewNode(int n1, int n2)
{
    node = new Node;
    node->x = n1;
    node->y = n2;
    if (!list_h)
        list_h = node;
    else
        list_t->next = node;
    list_t = node;
    list_t->next = NULL;
}
```

---

This function takes as parameters the values for the new node’s x and y members. First the new node is allocated on the heap, after which the x and y members are set to their appropriate values. Then, using the same code examined in Listing 33.3, the new node is added to the list.

To display the contents of the list, you can call this class’s DisplayList() function, which is shown in Listing 33.7.

---

**Listing 33.7 lst33\_07.cpp—The DisplayList() Member Function**

---

```
void List::DisplayList(void)
{
    node = list_h;
    while (node)
    {
        cout << node->x << ',' << node->y << '\n';
        node = node->next;
    }
}
```

---

This function simply scans the list (using the temporary node pointer, so it doesn’t destroy list\_h), printing the contents of x and y. Unlike the program in Listing 33.3, each node isn’t deleted after it’s printed. That job is left for the class’s destructor, shown in Listing 33.8.

---

**Listing 33.8 lst33\_08.cpp—The List Class’s Destructor**

---

```
List::~~List(void)
```

```

{
    while (list_h)
    {
        node = list_h;
        list_h = list_h->next;
        delete node;
    }
}

```

---

As with any class, the List class's destructor is called when a List object goes out of scope or when a dynamically allocated List object is deleted. The destructor then deletes every node in the list, using the same method that you saw in Listing 33.3 (but without printing the contents of the node before deleting it).

Listings 33.9 and 33.10 are the List class's implementation and the new main program, respectively.

### **Listing 33.9 lst33\_09.cpp—The List Class's Implementation**

---

```

#include "list.h"

List::List(void)
{
    list_h = list_t = NULL;
}

List::~List(void)
{
    while (list_h)
    {
        node = list_h;
        list_h = list_h->next;
        delete node;
    }
}

void List::MakeNewNode(int n1, int n2)
{
    node = new Node;
    node->x = n1;
    node->y = n2;
    if (!list_h)
        list_h = node;
    else
        list_t->next = node;
    list_t = node;
    list_t->next = NULL;
}

```

```

void List::DisplayList(void)
{
    node = list_h;
    while (node)
    {
        cout << node->x << ',' << node->y << '\n';
        node = node->next;
    }
}

```

---

### **Listing 33.10 lst33\_10.cpp—A Test Program for the List Class**

---

```

#include "list.h"

void main(void)
{
    List list;

    for (int i = 0; i < 10; ++i)
        list.MakeNewNode(i, i*10);
    list.DisplayList();
}

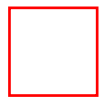
```

---

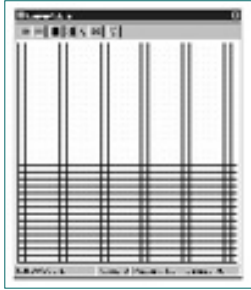
The main program is much shorter and clearer than the original program. Although the effort of creating the class for such a small program might or might not be worthwhile, imagine how much easier it would be to use a similar class in a large program that must handle multiple lists. By using a list class, you no longer must worry about initializing pointers or linking nodes. You don't even have to worry about releasing nodes from memory, because the class's destructor takes care of this task for you. By using the class, your main program is clean and to the point, uncluttered with the details of handling a linked list.

## **The Life Program**

You now know what linked lists are and how to handle them. You've even created a simple list class that demonstrates how you might go about using objected-oriented programming techniques to better organize programs that need to use linked lists. It's time to put your knowledge of linked lists to work, by examining the Life program, which uses MFC's more complicated and complete linked list class, called CPtrList. This program's listing will be explored a piece at a time, in the order in which it is executed. But first, run the program and see what it does. You can find the executable file, as well as all of the source code, in the Chap33\LIFE folder on this book's CD-ROM.



When you run Life, the main screen appears, as shown in Figure 33.7. Most of the screen is made up of the grid in which your cells will live and die. Above the grid is the toolbar, which contains several command buttons used to control the program. At the bottom of the screen is the status bar, where you can see the current speed setting for the simulation, the maximum number of generations for a run of the simulation, and the currently displayed generation. Before the simulation starts, the speed is set to 10, the maximum generations to 100, and the currently displayed generation to 0.



**FIG. 33.7** Life's main window features a toolbar and a status bar.

To get started, you must first seed the grid with cells. To do this, place your mouse pointer where you want to place a cell, and then click the left button. A red cell appears where you clicked. If you want to place cells quickly, click the Random toolbar button (the fourth button) or press F5 on your keyboard. Each time you choose the Random command, the program places more cells on the grid.

When you've placed your cells, activate the simulation by selecting the Start button, or by pressing F2. When you select Start, the simulation springs to life, with cells living and dying as they speed through their life cycles. To stop the simulation before the generations run out, click the Stop button or press F3.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Brief
 Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

**BROWSE**  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Right after the Stop button is the Clear button, which removes all of the cells from the grid. You can also select the Clear command by pressing F4. The Generation button (F6) sets the generation count. When you select this button, the Generations dialog box appears, as shown in Figure 33.8. To change the generation setting, type a number from 1 to 64,000.



**FIG. 33.8** The Generations dialog box sets the maximum number of generations.

You might want to view the simulation at slower speeds so that you can see more clearly the patterns that emerge from specific cell configurations. You can set the simulation to one of ten speeds by selecting the Speed button (F7). The Simulation Speed dialog box then appears, as shown in Figure 33.9. Enter a value from 1 to 10. (1 is the slowest, and 10 is the fastest.) Invalid entries yield the default value of 10.



**FIG. 33.9** The Speed dialog box enables you to run the simulation at one of ten speeds.

## Examining Life

Now that you know how to use the program, it’s time to examine the listing and see how Life works, especially how it handles its linked list using MFC’s CPtrList class. Listings 33.11 through 33.18 comprise the program’s source code.

### Listing 33.11 lifeapp.h—The Application Class’s Header File

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// LIFEAPP.H: Header file for the CLifeApp class, which
//               represents the application object.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

class CLifeApp : public CWinApp
{
public:
    CLifeApp();

    // Virtual function overrides.
    BOOL InitInstance();
};
```

### Listing 33.12 lifeapp.cpp—The Application Class’s Implementation File

```

////////////////////////////////////
// LIFEAPP.CPP: Implementation file for the CLifeApp,
//           class, which represents the application
//           object.
////////////////////////////////////

#include <afxwin.h>
#include "lifeapp.h"
#include "mainfrm.h"

// Global application object.
CLifeApp LifeApp;

////////////////////////////////////
// Construction/Destruction.
////////////////////////////////////
CLifeApp::CLifeApp()
{
}

////////////////////////////////////
// Overrides
////////////////////////////////////
BOOL CLifeApp::InitInstance()
{
    m_pMainWnd = new CMainFrame();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

```

---

### Listing 33.13 mainfrm.h—The Main Window Class's Header File

---

```

////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//           represents the application's main window.
////////////////////////////////////

#include <afxext.h>

const SQUARESIZE = 12;
const NUMBEROFROWS = 32;
const NUMBEROFCOLS = 32;
const VEROFFSET = 34;
const HOROFFSET = 4;
const ALIVE = 1;
const DEAD = 0;

class CMainFrame : public CFrameWnd
{
    // Protected data members.

```

```

protected:
    CToolBar m_toolBar;
    CStatusBar m_statusBar;
    CBitmap* m_pBitmap;
    BOOL m_world[NUMBEROFROWS][NUMBEROFCOLS];
    UINT m_nbrs[NUMBEROFROWS][NUMBEROFCOLS];
    UINT m_curGeneration;
    UINT m_generations;
    UINT m_speed;
    CPtrList* m_pLive;
    CPtrList* m_pDie;
    CPtrList* m_pNextLive;
    CPtrList* m_pNextDie;

// Constructor and destructor.
public:
    CMainFrame();
    ~CMainFrame();

// Overrides.
protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Message map functions.
protected:
    afx_msg void OnPaint();
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnDestroy();

    afx_msg void OnStart();
    afx_msg void OnStop();
    afx_msg void OnClear();
    afx_msg void OnRandomCells();
    afx_msg void OnGenerations();
    afx_msg void OnSpeed();
    afx_msg void OnAbout();

// Update command UI handlers.
    afx_msg void OnUpdateStartUI(CCmdUI* pCmdUI);
    afx_msg void OnUpdateStopUI(CCmdUI* pCmdUI);
    afx_msg void OnUpdateSpeedUI(CCmdUI* pCmdUI);
    afx_msg void OnUpdateGenerationsUI(CCmdUI* pCmdUI);
    afx_msg void OnUpdateClearUI(CCmdUI* pCmdUI);
    afx_msg void OnUpdateRandomCellsUI(CCmdUI* pCmdUI);

// Protected member functions.
protected:
    void ClearBitmap();
    void DrawGrid();
    BOOL ClickInsideGrid(CPoint point);
    void DrawCell(UINT col, UINT row, BOOL alive);

```

```
void CreateLists();
void ReleaseNodes(CPtrList* pList);
void Live();
void Die();
void AddNbrs();
void SubNbrs();
void CalcLimits(int c, int r, int &pxlow, int &pxhigh,
                int &pylow, int &pyhigh);
void TransferList(CPtrList** pDestList, CPtrList** pSrcList);

DECLARE_MESSAGE_MAP()
};
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

### Listing 33.15 gendlg.h—The Generation Dialog Box’s Header File

```

////////////////////////////////////
// GENDLG.H: Header file for the CGenDlg class.
////////////////////////////////////

class CGenDlg : public CDialog
{
    // Constructor.
public:
    CGenDlg(CWnd* pParent);

    // Data transfer variables.
public:
    UINT m_generations;

    // Overrides.
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
};

```

### Listing 33.16 gendlg.cpp—The Generation Dialog Box’s Implementation File

```

////////////////////////////////////
// GENDLG.CPP: Implementation file for the CGenDlg class.
////////////////////////////////////

#include <afxwin.h>
#include "resource.h"
#include "gendlg.h"

////////////////////////////////////
// CONSTRUCTOR
////////////////////////////////////
CGenDlg::CGenDlg(CWnd* pParent) :
    CDialog(IDD_GENERATIONSDLG, pParent)
{
    // Initialize data transfer variables.
}

```

```

        m_generations = 100;
    }

    //////////////////////////////////////
    // Overrides.
    //////////////////////////////////////
void CGenDlg::DoDataExchange(CDataExchange* pDX)
{
    // Call the base class's version.
    CDialog::DoDataExchange(pDX);

    // Associate the data transfer variables with
    // the ID's of the controls.
    DDX_Text(pDX, IDC_GENERATIONS, m_generations);
    DDV_MinMaxUInt(pDX, m_generations, 1, 64000);
}

```

---

### **Listing 33.17 spddlg.h—The Speed Dialog Box's Header File**

---

```

    //////////////////////////////////////
    // SPDDL.G.H: Header file for the CSpeedDlg class.
    //////////////////////////////////////

class CSpeedDlg : public CDialog
{
    // Constructor.
public:
    CSpeedDlg(CWnd* pParent);

    // Data transfer variables.
public:
    UINT m_speed;

    // Overrides.
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
};

```

---

### **Listing 33.18 spddlg.cpp—The Speed Dialog Box's Implementation File**

---

```

    //////////////////////////////////////
    // SPDDL.G.CPP: Implementation file for the CSpeedDlg class.
    //////////////////////////////////////

#include <afxwin.h>
#include "resource.h"

```

```

#include "spddlg.h"

/////////////////////////////////////////////////////////////////
// CONSTRUCTOR
/////////////////////////////////////////////////////////////////
CSpeedDlg::CSpeedDlg(CWnd* pParent) :
    CDialog(IDD_SPEEDDLG, pParent)
{
    // Initialize data transfer variables.
    m_speed = 1;
}

/////////////////////////////////////////////////////////////////
// Overrides.
/////////////////////////////////////////////////////////////////
void CSpeedDlg::DoDataExchange(CDataExchange* pDX)
{
    // Call the base class's version.
    CDialog::DoDataExchange(pDX);

    // Associate the data transfer variables with
    // the ID's of the controls.
    DDX_Text(pDX, IDC_SPEED, m_speed);
    DDV_MinMaxUInt(pDX, m_speed, 1, 10);
}

```

---

## The Application and Dialog Box Classes

Most of the action in the Life program takes place in the CMainFrame class. However, you should take some time now to look over all of the listings that make up the program so that you get a good overview of all of the classes used. These last few chapters are designed to, not only provide you with new programming techniques, but also to reinforce and review the basic MFC programming techniques that you've developed over the course of reading this book.

Listings 33.11 and 33.12, for example, are the header and implementation files for the program's application class, CLifeApp. You should now fully understand why the class overrides the base class's InitInstance() virtual member function, as well as why the program must create its application object as a global object. If you're still a little fuzzy on the ins and outs of writing an application class, please refer back to Chapter 4, "Constructing an MFC Program from Scratch."

Skipping over the main window class, CMainFrame, for the time being, I will mention that the Life program also features two dialog box classes, one for the Generations dialog box and one for the Speed dialog box. These dialog box classes—CGenDlg and CSpeedDlg—are shown in listings 33.15 through 33.18. Notice that the classes are very similar, both providing data members for data that must be copied from the dialog box, as well as overriding the base class's DoDataExchange() virtual member function to transfer data to and from the dialog box. If none of this stuff rings a bell, march right over to Chapter 7, "Programming Dialog Boxes," to get the inside story.

## The Main Window Class

As I mentioned previously, the main window class, `CMainFrame`, is where the Life program gets its life. Start by examining the class's header file, as shown in Listing 33.13. At the top of the header file, you see some constants declared, as shown in Listing 33.19.

### Listing 33.19 `lst33_19.cpp`—The Life Application's Constants

---

```
const SQUARESIZE = 12;
const NUMBEROFROWS = 32;
const NUMBEROFCOLS = 32;
const VEROFFSET = 34;
const HOROFFSET = 4;
const ALIVE = 1;
const DEAD = 0;
```

---

Using these constants in the program, not only makes the program easier to modify, but also makes the program much easier to read.

Inside the class's declaration, the program declares a number of data members. The first two represent the program's toolbar and status bar objects, as shown here:

```
CToolBar m_toolBar;
CStatusBar m_statusBar;
```

Next, the class declares a pointer to a `CBitmap` object, like this:

```
CBitmap* m_pBitmap;
```

The bitmap whose address will be stored in this pointer holds the main window's graphical data that gets displayed each time the application receives a `WM_PAINT` message.

After the bitmap pointer, the class declares two arrays for storing information about the cells in the simulation, like this:

```
BOOL m_world[NUMBEROFROWS][NUMBEROFCOLS];
UINT m_nbrs[NUMBEROFROWS][NUMBEROFCOLS];
```

The `m_world[][]` array holds the status—living or dead—of the cells in the simulation's on-screen grid. A value of `ALIVE` in an array element means that that cell is living, whereas a value of `DEAD` means, of course, that the cell is not living. The `m_nbrs[][]` array holds the neighbor counts for each cell in the grid. For example, if cell 0,0 in the `m_world[][]` array has three living, adjacent cells, the value in element 0,0 of the `m_nbrs[][]` array will be 3.

The class then declares data members to hold status information for the simulation, like this:



```
UINT m_curGeneration;  
UINT m_generations;  
UINT m_speed;
```

The m\_curGeneration variable holds the number of the current generation of cells on the screen, whereas m\_generations holds the setting for the maximum generations, and m\_speed holds the currently selected simulation speed.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

Finally, the class declares pointers to CPtrList objects, which are linked lists of pointers. Each pointer in one of these lists will hold the address of the data for a single cell. The m\_pLive and m\_pDie lists hold the cells that will live and die in a given generation. The m\_pNextLive and m\_pNextDie lists are used as temporary storage for cells that are eventually transferred to the live and die lists. You'll see how this works when you get further into the program's code.

The CMainFrame class's header file also lists the class's member functions. The first of these is the overridden PreCreateWindow() function:

```
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

As you should already know, in PreCreateWindow(), the program can modify the application's main window before it's displayed.

Next are the message map functions, which respond to, not only Windows messages, but also messages generated by the toolbar's buttons and by MFC's command-UI system. The message map functions for system messages are listed first, as shown in Listing 33.20.

#### Listing 33.20 lst33\_20.cpp—Message Map Functions for System Messages

```
afx_msg void OnPaint();
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnTimer(UINT nIDEvent);
afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg void OnDestroy();
```

After the system message map functions come the response functions for the toolbar's buttons (see Listing 33.21).

#### Listing 33.21 lst33\_21.cpp—Response Functions for the Toolbar Buttons

```
afx_msg void OnStart();
afx_msg void OnStop();
afx_msg void OnClear();
afx_msg void OnRandomCells();
afx_msg void OnGenerations();
afx_msg void OnSpeed();
afx_msg void OnAbout();
```

The last message map functions listed (see Listing 33.22) handle the command-UI messages.

#### Listing 33.22 lst33\_22.cpp—The Command-UI Response Functions

```
afx_msg void OnUpdateStartUI(CCmdUI* pCmdUI);
afx_msg void OnUpdateStopUI(CCmdUI* pCmdUI);
```

```
afx_msg void OnUpdateSpeedUI(CCmdUI* pCmdUI);
afx_msg void OnUpdateGenerationsUI(CCmdUI* pCmdUI);
afx_msg void OnUpdateClearUI(CCmdUI* pCmdUI);
afx_msg void OnUpdateRandomCellsUI(CCmdUI* pCmdUI);
```

---

The command-UI functions control how the toolbar buttons look and act. If you don't remember about command-UI functions, turn back to Chapter 4, "Constructing an MFC Program from Scratch," and Chapter 6, "Using Menus."

- See "Responding to Windows Messages," p. 68
- See "Defining the Message Map," p. 110

Finally, the CMainFrame class has many protected member functions that control the simulation (see Listing 33.23).

#### **Listing 33.23** `lst33_23.cpp`—The Class's Protected Member Functions

---

```
void ClearBitmap();
void DrawGrid();
BOOL ClickInsideGrid(CPoint point);
void DrawCell(UINT col, UINT row, BOOL alive);
void CreateLists();
void ReleaseNodes(CPtrList* pList);
void Live();
void Die();
void AddNbrs();
void SubNbrs();
void CalcLimits(int c, int r, int &ampxlow, int &ampxhigh,
               int &ampylow, int &ampyhigh);
void TransferList(CPtrList** pDestList, CPtrList** pSrcList);
```

---

You will see what all these functions do as you dig deeper into the program—which you'll do in the very next section.

### **Creating the Window and Initializing Variables**

Now you're ready to see what makes this program tick. As you know, when the application object is created, the program's main window is also created, in the CLifeApp class's `InitInstance()` function. The main window's constructor first creates the window:

```
Create(NULL, "Conway's Life",
       WS_OVERLAPPED | WS_SYSMENU | WS_CLIPCHILDREN);
```

The constructor also initializes the `m_world[][]` array to all dead cells:

```
for (UINT row=0; row<NUMBEROFROWS; ++row)
    for (UINT col=0; col<NUMBEROFCOLS; ++col)
        m_world[row][col] = DEAD;
```

It then initializes the simulation's status variables, like this:

```
m_generations = 100;
m_curGeneration = 0;
m_speed = 10;
```

Finally, the constructor creates the program's linked lists, as shown in Listing 33.24.

```
m_pLive = new CPtrList;  
m_pDie = new CPtrList;  
m_pNextLive = new CPtrList;  
m_pNextDie = new CPtrList;
```

---

## The Toolbar and Status Bar

In this program, the toolbar is created from the old `CToolBar` class rather than from the `CToolBarCtrl` class, which can be used only with Windows 95 or Windows NT. The `CToolBar` class isn't as fancy as `CToolBarCtrl`, but it can definitely get the job done. The same thing can be said for the `CStatusBar` control—which is the basis for the Life application's status bar—and its relationship with the Windows 95 and NT `CStatusBarCtrl` class. For more information on the `CToolBarCtrl` and `CStatusBarCtrl` classes, see Chapter 11, "Toolbars and Status Bars."

The `OnCreate()` function, which responds to Windows' `WM_CREATE` message, is where the program creates its toolbar and status bar. First `OnCreate()` calls the base class's `OnCreate()` function to ensure that the base class gets to do its thing:

```
CFrameWnd::OnCreate(lpCreateStruct);
```

Then the function creates the toolbar object, like this:

```
m_toolBar.Create(this);
```

The constructor's single argument is a pointer to the parent window.

Next the program loads the toolbar from the resource file.

```
m_toolBar.LoadToolBar(IDR_TOOLBAR1);
```

The `LoadToolBar()` member function takes as its single argument the toolbar resource ID. Obviously, you must create the toolbar resource ahead of time, using Developer Studio's toolbar editor. (You can also find the resources on this book's CD-ROM, in the Chap30\Life folder.) Creating the resource is a simple matter of drawing the icons for the buttons and giving the buttons their IDs. Consult your Visual C++ online documentation for more information on constructing toolbars with the toolbar editor.

The last thing the `OnCreate()` function does with the toolbar is set its style to display ToolTips, like this:

```
??? Author, I made "tool tips" "ToolTips," per Que's convention; OK?--  
m_toolBar.SetBarStyle(m_toolBar.GetBarStyle() |  
    CBRS_TOOLTIPS | CBRS_FLYBY);
```

The nested call to the `GetBarStyle()` member function retrieves the toolbar's current style. The program ORs this style with the style constants `CBRS_TOOLTIPS`, which allows the toolbar to display ToolTips, and `CBRS_FLYBY`; this, in turn, enables the status bar to display message text for the ToolTip at the same time that the ToolTip is displayed. Where do the ToolTips and message text come from? When you're constructing your toolbar with the toolbar editor, double-click an icon in the toolbar to display the button's Toolbar Button Properties property sheet. In the prompt line, type the button's message text and ToolTip, separated by the newline control character (`\n`).

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

Creating the status bar is just a little bit trickier, except for the fact that you don't have to create a resource for the status bar. In the Life application, the program first creates an array containing IDs with which panels in the status bar are associated, as shown in Listing 33.25.

### Listing 33.25 lst33\_25.cpp—Defining the Structure for the Status Bar's Panels

```
static UINT indicators[] =
{
    ID_SEPARATOR,
    ID_INDICATOR_SPEED,
    ID_INDICATOR_GENERATIONS,
    ID_INDICATOR_CURGENERATION
};
```

Each of the IDs in the indicators[] array is the ID of a string resource in a string table. You have to create this string resource in your resource file, matching up the strings that you want displayed in the status bar with their IDs. When your program is running, MFC takes care of displaying the correct strings in the status bar's panels. The ID\_SEPARATOR ID in the array saves a space for the panel that will display the toolbar's ToolTip messages.

After defining the array of IDs, the program calls the status bar's SetIndicators() member function to tell MFC what panels the program needs, like this:

```
m_statusBar.SetIndicators(indicators, 4);
```

This function's two arguments are the address of the array containing the string IDs and the number of IDs in the array.

Next, the program sets the first pane's style, like this:

```
m_statusBar.SetPaneInfo(0, m_statusBar.GetItemID(0),
    SBPS_STRETCH, 0);
```

The SetPaneInfo() function's four arguments are the pane's zero-based index (that is, 0 is the index of the first pane), the pane's new ID, the pane's new style, and the pane's new width. In this case, the program wants only to set the style, so the nested call to GetItemID() returns the pane's current ID. The SBPS\_STRETCH style sets the first pane so that it stretches to fill all remaining space in the status bar. Because of this style, the pane's new width, which is SetPaneInfo()'s last argument, can be set to 0. Other styles

you can use are SBPS\_NOBORDERS, SBPS\_POPOUT, SBPS\_DISABLED, and SBPS\_NORMAL.

Finally, three calls to the status bar's SetPaneText() function set the strings in the panes exactly as the program wants them:

```
m_statusBar.SetPaneText(1, "Speed: 10", TRUE);  
m_statusBar.SetPaneText(2, "Max Gens: 100", TRUE);  
m_statusBar.SetPaneText(3, "Generation: 0", TRUE);
```

The SetPaneText() function's three arguments are the pane's index, the new text, and a Boolean value indicating whether the pane should be redrawn immediately with the new text. Why set the pane's text when the program's resource file already included strings for the status bar? In this program's case, the strings in the string-table resource are used only to reserve the right amount of space in each pane. The strings themselves will change continuously as the user uses the program, changing the options and running the simulation.

## The Window's Bitmap

Because of the nature of the user's interaction with the program, the Life application uses an in-memory bitmap to store the current image being displayed in the window. Life's main window (CMainFrame) uses its OnPaint() function to transfer a bitmap from memory to the screen. Other than that bitmap transfer, OnPaint() does no other window painting.

Obviously, the bitmap that OnPaint() displays must be modified somewhere in the program. These modifications occur whenever the user places a new cell on the grid. Later in this section, you'll see what happens in the program when the user adds a cell to the world grid. For now, this discussion concentrates on the OnCreate() function, which creates the in-memory bitmap at the start of the program run.

In the previous section, you saw how OnCreate() gives the window its toolbar and status bar. The last few lines take care of the bitmap. First the program creates a device context (DC) for the window's client area:

```
CClientDC clientDC(this);
```

Then the program creates a new bitmap object and makes the bitmap compatible with the window's client DC:

```
m_pBitmap = new CBitmap;  
m_pBitmap->CreateCompatibleBitmap(&clientDC, 640, 480);
```

The CreateCompatibleBitmap() function requires three arguments: the DC with which the bitmap should be compatible and the width and height of the bitmap.

To draw the starting image on the bitmap, OnCreate() calls the ClearBitmap() and DrawGrid() member functions. The ClearBitmap() function clears the bitmap to a white rectangle. To do this, the program creates a DC for the window's client area, and that creates a memory DC that's compatible with the window, like this:

```
CClientDC dc(this);
CDC memDC;
memDC.CreateCompatibleDC(&amp;dc);
```

Because the bitmap is compatible with the window DC and the memory DC is compatible with the window DC, the bitmap can be selected into the memory DC, like this:

```
memDC.SelectObject(m_pBitmap);
```

By selecting the bitmap into the DC, the program can draw on the bitmap (just as if the bitmap were any other type of display device), which it does by creating a white brush and using the brush in a call to the FillRect() function. The FillRect() function then fills the entire bitmap with white:

```
CBrush brush( RGB(255,255,255) );
memDC.FillRect( CRect(0, 0, 639, 479), &brush );
```

As you might have guessed, the DrawGrid() function paints the gridlines on the bitmap. In that function, the program creates a memory DC and selects the bitmap into it, just as was described for ClearBitmap(). With the DC in hand, the program can draw the gridlines on the bitmap in memory. The first for loop draws the grid's vertical lines, as shown in Listing 33.26.

---

#### **Listing 33.26 lst33\_26.cpp—Drawing the Vertical Gridlines**

---

```
for (int x=0; x<=NUMBEROFCOLS; ++x)
{
    memDC.MoveTo( SQUARESIZE*x+HOROFFSET, VEROFFSET );
    memDC.LineTo( SQUARESIZE*x+HOROFFSET,
                  SQUARESIZE*NUMBEROFROWS+VEROFFSET + 1 );
}
```

---

In a similar manner, DrawGrid() also draws the horizontal gridlines, as seen in Listing 33.27.

---

#### **Listing 33.27 lst33\_27.cpp—Drawing the Horizontal Gridlines**

---

```
for (int y=0; y<=NUMBEROFROWS; ++y)
{
    memDC.MoveTo( HOROFFSET, SQUARESIZE*y+VEROFFSET );
    memDC.LineTo( SQUARESIZE*NUMBEROFCOLS+HOROFFSET,
                  SQUARESIZE*y+VEROFFSET );
}
```

---

As I said previously, the main window class's OnPaint() function ordinarily takes care of every case of window repainting. In Life, however, trying to update the screen in OnPaint() using the raw data stored in the m\_world[][] array makes the program look clunky. This is because it takes a couple of seconds to read through the array and to



paint each cell found there onto the on-screen grid. Painting a bitmap in the window is much faster.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

So when the user clicks the mouse inside the grid, the OnLButtonDown() function takes care of the drawing tasks in an unusual way, painting the selected cell both on the screen and on the in-memory bitmap. Because the program keeps the bitmap up-to-date, when the program needs to update a portion of its window, it can just copy the bitmap rather than updating the window by reading through the m\_world[][] array.

If you look at OnLButtonDown(), you'll see that it first checks the value of m\_curGeneration, like this:

```
if (m_curGeneration != 0)
    return;
```

The variable m\_curGeneration holds the number of the generation of cells displayed on the screen. This value is nonzero only when the simulation is actually running (after the user selects the Go command). Because the program doesn't let the user add cells while the simulation is going, the preceding lines cause an immediate return from OnLButtonDown() when m\_curGeneration is not 0.

If the simulation is not active, the program must check whether the user actually clicked inside the grid rather than somewhere else in the window. It does this by calling the ClickInsideGrid() function, which returns TRUE if the click was inside the grid:

```
if (ClickInsideGrid(point))
```

If the click is valid, the program calculates the column and row of the cell that the user selected:

```
UINT gridCol = (point.x - HOROFFSET) / SQUARESIZE;
UINT gridRow = (point.y - VEROFFSET) / SQUARESIZE;
```

Then the program checks whether the clicked cell is dead:

```
if (m_world[gridRow][gridCol] == DEAD)
```

The program checks the cell's status because there's no point in bringing a living cell to life. If the cell is dead, the program brings it to life by adding it to the m\_world[][] array and drawing it on the screen and the bitmap:

```
m_world[gridRow][gridCol] = ALIVE;
DrawCell(gridCol, gridRow, TRUE);
```

The DrawGrid() function must draw the cell on the screen and on the in-memory bitmap, so its first task is to create both a window and memory DC, like this:

```
CClientDC clientDC(this);
CDC memDC;
memDC.CreateCompatibleDC(&clientDC);
```

Next, DrawGrid() selects the bitmap into the memory DC so the program can draw on the bitmap:

```
memDC.SelectObject(m_pBitmap);
```

Then the function must determine whether it needs to create a white brush (to erase dead cells) or a red brush (to draw living cells). It does this by checking the alive flag, which is passed as the function's last argument. If alive is TRUE, the function must draw a living cell; otherwise, it must erase the cell. A simple if statement takes care of either eventuality, as shown in Listing 33.28.

---

**Listing 33.28 lst33\_28.cpp—Creating the Correct Brush for the Cell**

---

```
if (alive)
    pBrush = new CBrush(RGB(255,0,0));
else
    pBrush = new CBrush(RGB(255, 255, 255));
```

---

As you know, to use the brush, it must first be selected into the DC. In this case, two DCs—the window DC and the memory DC—need the brush, so the program selects the brush into both:

```
CBrush* pOldBrush1 = memDC.SelectObject(pBrush);
CBrush* pOldBrush2 = clientDC.SelectObject(pBrush);
```

Now that everything's ready to draw on the DCs, the program calculates the pixel coordinates at which to draw the cell:

```
UINT drawCol = col * SQUARESIZE + HOROFFSET;
UINT drawRow = row * SQUARESIZE + VEROFFSET;
```

A call to each DC's Rectangle() member function draws the new cell both in the window and on the bitmap, as you can see in Listing 33.29.

---

**Listing 33.29 lst33\_29.cpp—Drawing the New Cell**

---

```
memDC.Rectangle(drawCol, drawRow,
    drawCol + SQUARESIZE + 1, drawRow + SQUARESIZE + 1);
clientDC.Rectangle(drawCol, drawRow,
    drawCol + SQUARESIZE + 1, drawRow + SQUARESIZE + 1);
```

---

Finally, the program selects the old brushes back into the DCs, which frees up the created brush for deletion:

```
memDC.SelectObject(pOldBrush1);  
clientDC.SelectObject(pOldBrush2);  
delete pBrush;
```

## The Simulation's Main Loop

After the user has placed cells in the grid, he or she can select the Start command to put the simulation into action. When the user does this, the OnStart() member function gets the program's motor humming. It accomplishes this by creating the starting cell lists and setting a Windows timer, like this:

```
CreateLists();  
m_curGeneration = 1;  
SetTimer(1, 2100-m_speed*200, NULL);
```

As you can see, the timer's setting is based upon the current value of the m\_speed variable. But no matter what the selected speed, when the timer starts generating WM\_TIMER messages for the window, the OnTimer() function acts as the simulation's main loop, performing the same set of tasks again and again until the simulation ends.

Before starting in on OnTimer(), take a look at CreateLists(). This function is responsible for initializing the lists pointed to by m\_pLive and m\_pDie—the two linked lists that the simulation needs to get started—as well as initializing the starting neighbor counts. The function first calls ReleaseNodes() for each linked list, as shown in Listing 33.30.

### Listing 33.30 lst33\_30.cpp—Calling ReleaseNodes() for Each List

---

```
ReleaseNodes(m_pLive);  
ReleaseNodes(m_pDie);  
ReleaseNodes(m_pNextLive);  
ReleaseNodes(m_pNextDie);
```

---

ReleaseNodes() simply empties the given list.

---

#### Note:

When the program is first started, the lists are empty. But in subsequent calls to OnTimer(), your linked lists probably won't be empty because it is rare for every cell on the screen to be dead after the generations run out.

---

After clearing the lists, CreateLists() scans the newly created world array, creating a new node for each living cell in the array, as shown in Listing 33.31.

### Listing 33.31 lst33\_31.cpp—Initializing the List for the Living Cells

---

```
for (c=0; c<NUMBEROFCOLS; ++c)
    for (r=0; r<NUMBEROFROWS; ++r)
    {
        m_nbrs[r][c] = 0;
        if (m_world[r][c] == ALIVE)
        {
            pCell = new CPoint(c, r);
            m_pLive->AddTail(pCell);
        }
    }
}
```

---

As CreateLists() scans the world array, it also takes advantage of the loop to initialize all of the neighbor counts in the m\_nbrs[][] array to 0. Each cell in the Life program is represented by a CPoint object that holds the cell's coordinates in the grid. To add a cell to the linked list, the program first creates a CPoint object for the cell. Calling the list's AddTail() member function then adds the new cell to the end of the list.

After creating the linked list pointed to by m\_pLive, the CreateLists() function calls the AddNbrs() function, which updates the neighbor counts and creates the m\_pNextLive and m\_pNextDie list for cells that might (or might not) live and die in the next generation.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 34

# Understanding Recursion

- Learn how recursion can simplify programming problems
- Study useful examples of recursion
- Discover how to avoid stack problems when using recursion
- Learn about trees and tree traversals
- Use recursion to write a familiar game program

During your programming career, you've probably heard the words “divide and conquer” quite often. This is because experienced programmers know that writing a large program can be a psychologically draining challenge. When you think about all that goes into a full-length program, it's easy to become overwhelmed by the magnitude of the job. So, just as you read a book page by page or clean a house room by room, you write a program one function at a time. In this way, you can understand a huge task that might otherwise be beyond your abilities to grasp as a whole.

You can adopt the divide-and-conquer strategy in several ways, including object-oriented programming and structured programming. **Recursion**, the subject of this chapter, is another technique you can use to break complex tasks into their components. Using recursion, you can take a repetitive task and reduce it to a single step that is repeated again and again until you obtain the desired result.

In this chapter, you learn what recursion is and how it can be used to replace complex code with short and elegant functions. You also use recursion in a full-length program (a game!), applying what you've learned to a practical case.

## Recursion: Barrels Within Barrels

When I was a kid, one of my favorite toys was a bunch of nested plastic barrels. You'd unscrew the first barrel, only to find a smaller one inside. In that barrel was yet another smaller barrel, and so on. Finally, in the tiniest barrel, was a small plastic rabbit. I spent hours fascinated with those barrels.

Recursion fascinates me in the same way, probably because recursion is a lot like those nested barrels. With recursion, you work your way deeper and deeper into an operation until you finally find that little bunny—the result of

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

the operation you're trying to perform. Using recursion, complex operations can be programmed with only a few lines of code.

But what exactly is recursion? In a program, recursion occurs when a function calls itself. This might sound a little crazy. Why would a function want to call itself? When you called a function in past programs, you expected the function to do its job and to return. With recursion, though, you must think about functions differently. Rather than finishing a job, a recursive function does only a small portion of the task and then passes what's left to another call to itself.

The simplest recursive function looks something like Listing 34.1:

---

**Listing 34.1 lst34\_01.cpp—The Simplest Recursive Function**

---

```
void Recursive(void)
{
    Recursive();
}
```

---

This function accomplishes nothing. Worse, it's an infinite loop. The Recursive() function is called again and again, until, finally, you run out of stack space. For a recursive function to operate correctly, it needs some way to break out of the recursion. A more useful template for a recursive function is shown in Listing 34.2:

---

**Listing 34.2 lst34\_02.cpp—A More Useful Recursive-Function Template**

---

```
void Recursive(void)
{
    if (condition)
        return;
    else
        Recursive();
}
```

---

Here, when condition equals true, you return immediately rather than call Recursive() again. After breaking out of the recursion, previous calls to Recursive()—all of which have already executed their last statement (the else statement)—also return, until you finally return to the original call to Recursive().

This last example function doesn't accomplish much, but it does illustrate the most important elements of a recursive function, as follows:

- A recursive function calls itself.
- A recursive function must contain a conditional statement that breaks the recursive cycle.

## A Real-World Example of Recursion

Now, how about an example that does something? If you consider those little barrels mentioned in the last section, you might see how recursion can simplify a programming task. To get to the bunny in those barrels, a child must open barrels, one after another, until she or he reaches the last one. Opening a single barrel is only a small part of the entire task. After the first barrel is opened, the same function must be performed on each remaining barrel.

Think of barrel-opening as a function in a program. In fact, you can do more than think about it; you can learn how to write it. Listing 34.3 is a program that simulates the bunny-in-barrels toy.

The complete source code and executable file for the Barrels application can be found in the CHAP34\Barrels directory on this book's CD-ROM.



### Listing 34.3 Barrels.cpp—The Bunny-in-Barrels Program

---

```
#include <iostream.h>
#include <conio.h>

void OpenBarrel(int num)
{
    if (num == 0)
        cout << "Got the bunny!" << endl;
    else
    {
        cout << "Opening barrel #" << num << endl;
        OpenBarrel(num-1);
    }
}

void main(void)
{
    OpenBarrel(10);
    getch();
}
```

---

When you compile and run the program, you see the following output:

```
Opening barrel #10
Opening barrel #9
Opening barrel #8
Opening barrel #7
Opening barrel #6
Opening barrel #5
Opening barrel #4
Opening barrel #3
Opening barrel #2
```



Opening barrel #1  
Got the bunny!

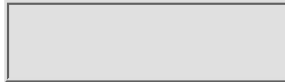
<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

In the main program, `OpenBarrel()` is called with a parameter of 10. (This parameter indicates the number of barrels and can be any integer.) In `OpenBarrel()`, the value of `num` is first checked to see whether the program has reached the last barrel. If it has, it prints the Got the bunny! message. Otherwise, the program calls `OpenBarrel()` again with a parameter of `num-1`. This call invokes `OpenBarrel()` a second time—before the first invocation has ended—with a value of 9. This second invocation checks the value of `num`, finds it to be 9, and calls `OpenBarrel()` a third time, this time with a value of 8. This process continues until a call to `OpenBarrel()` gets a value of 0 for `num`, triggering the program to print Got the bunny!

#### Note:

Note that each invocation of `OpenBarrel()` has its own `num` variable. The `num` variable for the first invocation is not the same `num` you use in the second invocation. This is important to understand because this series of values eventually breaks the program out of the recursion.

## A Power Function Using Recursion

Although the bunny-in-barrels program illustrates recursion well by simulating an easy-to-grasp, real-world problem, it doesn't show how you might use recursion in your programs. It's unlikely that you'll ever need to write programs about bunnies and barrels. So take a look at a recursive function that accomplishes something worthwhile, but that is still as easy to understand as the barrel program.

Consider the value  $10^3$ . The result of the exponentiation is calculated by multiplying 10 by itself three times:  $10 \times 10 \times 10$ . You can use a for loop to calculate this value, but that's much too pedestrian for power programmers. Instead, you can perform this multiplication operation recursively. Listing 34.4 includes a recursive function, `Power()`, which calculates the value of any integer raised to a positive integer exponent.



The complete source code and executable file for the Power application can be found in the CHAP34\Power directory on this book's CD-ROM.

### Listing 34.4 Power.cpp—A Recursive Exponentiation Example

```
#include <iostream.h>
```

```
#include <conio.h>

int Power(int num, int exp)
{
    if (exp == 1)
        return num;
    else
        return num * Power(num, exp-1);
}

void main(void)
{
    cout << Power(10,3) << endl;
    getch();
}
```

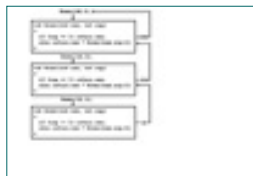
---

Examine this short program carefully. Although the `Power()` function is only a few lines long, a lot more is going on than might at first be apparent. Basically, this function calls itself repeatedly with smaller and smaller values of `exp`, until `exp` equals 1. At this point, instead of calling itself again, `Power()` simply returns the value `num`. (In the  $10^3$  example, `num` equals 10.) Notice that no calculations are performed until the recursion has occurred as many times as possible. Then it returns 10 to the previous invocation of `Power()`, which multiplies that return value by 10. The result (100) is passed on to the first invocation, which also multiplies it by 10, giving the final result of 1000.

Confused? Figure 34.1 will help dispel the mystery. Starting at the top of the figure, `Power()` is called with the parameters 10 and 3. In the first call to `Power()`, the `if` statement examines `exp` and finds it to be 3, so the `else` statement executes. In the `else` statement, `num` is multiplied by the value returned from `Power(num, exp-1)`.

The function can't perform the multiplication until it gets a return value from `Power()`, however, so it drops down to the second call to `Power()`, which gets the parameters 10 and 2. Again, the `if` statement is evaluated and program execution drops down to the `else` statement, which multiplies `num` by yet another call to `Power()`, this time with the parameters 10 and 1.

This brings the program to the third call to `Power()`, shown in the bottom box. This call gets the parameters 10 and 1. This time the `if` statement finds that `exp` is 1, so it immediately returns the value of `num`, which in this case is 10.



**FIG. 34.1** Here's how  $10^3$  is solved recursively.

Notice that the program has performed no multiplication operations, because it has had no result from `Power()`. Instead, it has simply called `Power()` `exp` times. The multiplication takes place as the program works its way back out of the

recursions. The third recursion returns 10 to the second recursion, where this 10 is multiplied by num. The result of 100 is returned to the first call to Power(), which also multiplies the result by num. The result of 1000 is finally returned to your original call.

## Recursion and the Stack

As you shall soon see, recursion is useful for more than solving simple mathematical problems. Recursion is also used in sorting, tree-traversal, parsing and solving complex mathematical expressions, managing disk directories, and much more. In this chapter, you examine a tree-traversal routine. But first you have to know how recursive routines affect the stack and how this can get you into trouble.

Earlier in this chapter, in the section “Recursion: Barrels Within Barrels,” you looked at a simple recursive function that ran endlessly because there was no way to break out of the recursion. This function called itself repeatedly until it ran out of stack space. What does the stack have to do with recursion?

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Every time a function is called, certain values are placed on the stack. These values are part of something called a ***stack frame***. They include the parameters being passed to the particular function and the address to which the program should return after the function ends. The stack has only a limited amount of space, so it can hold only so many stack frames. When a recursive function calls itself too often, the stack fills with stack frames, until no space is left. And when the stack overflows, the program drops dead.

Listing 34.5 is a program that calls a recursive function containing no conditional with which it can break out of the recursion. Each invocation of the Recursive() function prints the call number on-screen, so you can see that the program is actually doing something.



The complete source code and executable file for the Stack1 application can be found in the CHAP34\Stack1 directory on this book's CD-ROM.

### Listing 34.5 STACK1.CPP—Version 1 of the Stack Overflow Program

```
#include <iostream.h>

void Recursive(int c)
{
    cout << c << ' ';
    Recursive(c+1);
}

void main(void)
{
    Recursive(1);
}
```

If you run this program under DOS, you see that it calls Recursive() approximately 8,000 times before it runs out of stack space. (If you run the program in a DOS box under Windows 95, it'll go a long, long time before it runs out of stack place, thanks to the huge amount of virtual memory Windows 95 allots a running program.)

Listing 34.6 is the same program, only this time the Recursive() function takes three parameters instead of one. By having more parameters, a call to Recursive() generates larger stack frames. Each call uses more stack space, so

this program can call the function only about 5,000 times (when running under DOS) before it runs out of stack space.

The complete source code and executable file for the Stack2 application can be found in the CHAP34\Stack2 directory on this book's CD-ROM.



---

**Listing 34.6 STACK2.CPP—Version 2 of the Stack Overflow Program**

---

```
#include <iostream.h>

void Recursive(int c, int c2, int c3)
{
    cout << c << ' ' ;
    Recursive(c+1, c2, c3);
}

void main(void)
{
    Recursive(1, 1, 1);
}
```

---

As you already know, every recursive function needs a conditional statement that eventually ends the recursion, something Listings 34.5 and 34.6 are missing. Listing 34.7 adds such a conditional statement that allows only 5,000 recursions.

The complete source code and executable file for the Stack3 application can be found in the CHAP34\Stack3 directory on this book's CD-ROM.



---

**Listing 34.7 STACK3.CPP—Version 3 of the Stack Overflow Program**

---

```
#include <iostream.h>

void Recursive(int c, int c2, int c3)
{
    cout << c << ' ' ;
    if (c==5000)
        return;
    Recursive(c+1, c2, c3);
}

void main(void)
{
    Recursive(1, 1, 1);
}
```

---

Does this solve your stack problem? Yes and no. As long as you don't change

the size of the stack or add additional parameters to the Recursive() function, you should have no trouble with the stack. Listing 34.8 shows that adding even a single integer parameter can get you into trouble, by overflowing the stack.

The complete source code and executable file for the Stack4 application can be found in the CHAP34\Stack4 directory on this book's CD-ROM.



---

### Listing 34.8 STACK4.CPP—Version 4 of the Stack Overflow Program

---

```
#include <iostream.h>

void Recursive(int c, int c2, int c3, int c4)
{
    cout << c << ' ';
    if (c>5000)
        return;
    Recursive(c+1, c2, c3, c4);
}

void main(void)
{
    Recursive(1, 1, 1, 1);
}
```

---

---

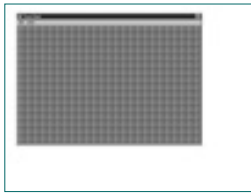
#### Caution:

Always be aware that you place a lot of data on the stack when using recursive routines. Moreover, the more parameters required by the recursive routines, the fewer number of stack frames that fit on the stack, limiting even further the number of recursive calls you can make. To avoid stack problems, recursive functions should use as few parameters as possible. Be especially careful of passing large data structures, such as arrays, as parameters in a recursive function. If you need to use a large data structure as a parameter to a recursive function, pass it by reference (which passes only the data's address) not by value (which passes the contents of the entire data structure).

---

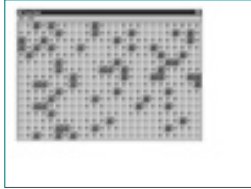
## An Example Application: Trap Hunt

That takes care of all the work. Now you can have a little fun. Listings 34.9 and 34.10 show the source code (only the application and main-window classes) for a puzzle game called *Trap Hunt*. When you compile and run the program, the main window appears. Choose the File, Start command to start a new game. When you do, the game screen appears with 400 buttons in a 25x16 grid (see Figure 34.2). To win the game, you must find the 60 traps hidden under these buttons.



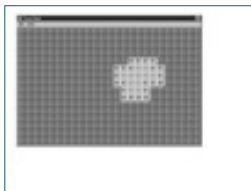
**FIG. 34.2** The game board contains 400 buttons.

Each square on the game board contains one of three things: a trap, a number, or a blank. To start, click a button on the game board with your left mouse button (***not*** the right button). If the button you choose reveals a trap, you lose the game (whew, that was fast!), and the entire game board is revealed (see Figure 34.3).



**FIG. 34.3** Clicking on a trap ends the game and displays the entire game board.

If the button reveals a blank square, every blank square connected to it is shown, up to and including bordering number squares (Figure 34.4). If a button reveals a number, it informs you of the number of traps adjacent to the selected button.



**FIG. 34.4** Clicking a button that hides a blank square clears a chunk of the board and gives you a lot of additional number clues.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[Brief](#)   [Full](#)  
[Advanced Search](#)  
[Search Tips](#)



[an error occurred while processing this directive]

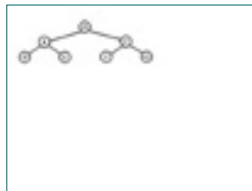
[Previous](#)
[Table of Contents](#)
[Next](#)

## Programming with Trees

The Trap Hunt program contains an excellent example of recursion that you can study to get further insight into this handy and interesting programming technique. In a previous discussion of ways to use recursion, tree-traversal routines were mentioned. This is the type of recursion used in Trap Hunt.

What's a *tree*? A tree is a data structure that connects a collection of items, called *nodes*. A tree starts with a root node. Connected to the root are any number of child nodes. Each child node, too, can have any number of its own child nodes. This hierarchy continues down the tree until a child node has no children of its own.

Figure 34.6 shows a general binary tree (that is, this tree has nothing to do with the TrapHunt program), which is a special type of tree that has left and right children for every node except the base nodes. Node A is the root node. Nodes B and C, which are called siblings because they are on the same level of the tree, are A's child nodes. Nodes B and C also have two child nodes each, the base nodes D, E, F, and G.



**FIG. 34.6** Each node of a binary tree has exactly two child nodes.

Recursion is particularly useful for traversing trees—that is, for following every path in the tree from the root to the base nodes. Listing 34.11 is a DOS program that creates and traverses the binary tree shown in Figure 34.6. The program's output follows:

```

At node A
At node B
At node D
At node E
At node C
At node F
At node G

```



The complete source code and executable file for the Tree application can be found in the CHAP34\Tree directory on this book's CD-ROM.

### Listing 34.11 TREE.CPP—Creating the Binary Tree Shown in Figure 34.6

---

```
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>

struct Node
{
    char name;
    Node *left, *right;
};

Node *tree;

void AddNodes(Node *node, char c1, char c2);
void TraverseTree(Node *n);

void main(void)
{
    tree = new Node;
    tree->name = 'A';
    AddNodes(tree, 'B', 'C');
    AddNodes(tree->left, 'D', 'E');
    AddNodes(tree->right, 'F', 'G');
    TraverseTree(tree);
    delete tree;
    getch();
}

void AddNodes(Node *node, char c1, char c2)
{
    Node *n = new Node;
    n->name = c1;
    n->left = NULL;
    n->right = NULL;
    node->left = n;
    n = new Node;
    n->name = c2;
    n->left = NULL;
    n->right = NULL;
    node->right = n;
}

void TraverseTree(Node *n)
{
    cout << "At node " << n->name << endl;
    if (n->left)
    {
```

```

        TraverseTree(n->left);
        delete n->left;
    }
    if (n->right)
    {
        TraverseTree(n->right);
        delete n->right;
    }
}

```

---

The program implements a node as a struct containing the node's label and pointers to the node's left and right children. In `main()`, the program first creates the tree's root node, which is appropriately named `tree`. Then it calls the `AddNodes()` function to create two child nodes for `tree`. The program also calls `AddNodes()` (indirectly by way of the `tree->left` and `tree->right` pointers) for each of `tree`'s child nodes to create their own child nodes. The operations here are similar to those you learned when studying linked lists.

- **See “Linked Lists,” p. 712**

There should be no need to go into the details of the tree construction. What you must examine closely, though, is the recursive procedure that traverses the tree structure. In Listing 34.11, that function is `TraverseTree()`.

Here's how the recursion works:

1. The program calls `TraverseTree()` from `main()` with `tree`, which is a pointer to the tree's root node.
2. In `TraverseTree()`, the function first prints a message, showing which node it's currently examining. In this case, the node is A.
3. Then the function checks node A's left pointer. If it's not NULL, A has a left child, so the function calls `TraverseTree()` recursively to check that left child, which is B.
4. This call initiates a second invocation of `TraverseTree()`, in which a message for node B is printed and node B's left pointer is checked.
5. Because node B also has a left child, the program calls `TraverseTree()` yet again, this time for node D. In this third invocation of `TraverseTree()`, the program prints D's message and checks its left pointer.
6. D has no left child, so the program drops out of the first if and checks D's right pointer. D has no right child either. So, the third invocation of `TraverseTree()` ends, and the program is back to the second, where it last checked B's left pointer.
7. The program is now finished with B's left child, D, so it deletes it and checks B's right pointer, only to discover that it has a right child, E. This means the program must call `TraverseTree()` to examine node E.
8. Because node E, like node D, has no left or right children, the program promptly returns to B, where it deletes E and steps back to the first invocation of `TraverseTree()`.
9. The program had last checked A's left pointer, so it can delete that left child and move to A's right child, C.

**10.** The right side of the tree is traversed the same way that the left side was, by visiting C, F, and finally G.

**11.** At the end of the traversal, the program returns to A with all nodes examined and all nodes deleted, except the root node. The root node, tree, is deleted in main().

## Trap Hunt's Trees

Trap Hunt uses trees. How? When the player selects a blank square, the program must reveal all blank squares connected to it, as well as any number squares adjacent to blank squares. It does this by forming a tree and traversing the tree recursively.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

## Chapter 35

# Developing a Class

- Encapsulation, inheritance, and polymorphism
- Class organization
- Function and operator overloading
- Virtual functions
- Class design

Throughout this book, you've used the many classes that Microsoft created for MFC. Creating classes, however, is not a magical trick that only master programmers who work for Microsoft can accomplish. Any C++ programmer—including you—can create classes and incorporate them into his or her programs. If you've never created your own classes, however, you might find the process to be a bit mysterious. In this final chapter, you learn how to supplement MFC with your own custom-written classes.

## A Review of Object-Oriented Programming Techniques

Before you can start writing your own classes, you have to be comfortable with object-oriented programming techniques. Object-oriented programming enables you to think of program elements as objects. In the case of a window object, you don't need to know the details of how it works, nor do you need to know about the window's private data. You need to know only how to call the various functions that make the window operate. Think about a car. To drive a car, you don't have to know the details of how a car works. You need to know only how to drive it. What's going on under the hood is none of your business. (If you try to make it your business, plan to face an amused mechanic who will have to straighten out your mess!)

If this were all there were to object-oriented programming, you wouldn't have gained much over standard structured-programming techniques. After all, with structured programming, you can create "black box" routines, which a programmer could then use without knowing how they work. Obviously, there must be much more to object-oriented programming than just hiding the details of a process. In this section, you'll discover data encapsulation, inheritance, and polymorphism, the features that give object-oriented programming its true power.

### Encapsulation

One major difference between conventional procedural programming and object-oriented programming is a handy thing called **encapsulation**. Encapsulation enables you to hide both the data and the functions that act on that data inside the object. After you do this, you can control access to the data, forcing programs to retrieve or modify data only through the object's interface. In strict object-oriented design, an object's data is always private to the object. Other parts of a program should never have direct access to that data.

How is this data hiding different from a structured-programming approach? After all, you could always hide data inside of functions, just by making that data local to the function. A problem arises, however, when you want to make the data of one function available to other functions. The way to do this in a structured program is to make the data global to the program, which gives any function access to it. It seems that you could use another level of scope—one that would make your data global to the functions that need it—but still prevent other functions from gaining access. Encapsulation does just that.

BROWSE  
BY TOPIC

The best way to understand object-oriented programming is to compare a structured program to an object-oriented program. Let's extend the car-object metaphor by writing a program that simulates a car trip. The first version of the program (a pseudo-DOS, or console, program), shown in Listing 35.1, uses a typical structured design.



The complete source code and executable file for the Car1 application can be found in the Chap35\Car1 directory on this book's CD-ROM.

### **Listing 35.1 Car1.cpp—The First Version of the Car Program**

---

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#define HOME 10

void StartCar(void)
{
    cout << "Car started." << endl;
    getch();
}

int SteerCar(int destination, int &amposition)
{
    cout << "Driving..." << endl;
    getch();
    if (++position == destination) return 1;
    return 0;
}

void BrakeCar(void)
{
    cout << "Braking." << endl;
    getch();
}

void ReverseCar(int &amforward, int &amposition)
{
    if (forward)
    {
        cout << "Backing up." << endl;
        getch();
        --position;
        forward = 0;
    }
    else forward = 1;
}

void TurnOffCar(void)
{
    cout << "Turning off car." << endl;
    getch();
}

int FindObstacle(void)
{

```

```

        int r = rand() % 5;
        if (r) return 0;
        return 1;
    }

    int position = 0, destination = HOME;
    int at_destination = 0;
    int obstacle, forward = 1;

    void main()
    {
        srand((unsigned)time(NULL));
        StartCar();
        while (!at_destination)
        {
            at_destination = SteerCar(destination, position);
            obstacle = FindObstacle();
            if (obstacle && !at_destination)
            {
                cout << "Look out! There's something in the road!" << endl;
                getch();
                BrakeCar();
                ReverseCar(forward, position);
                ReverseCar(forward, position);
            }
        }
        cout << "Ah, home at last." << endl;
        TurnOffCar();
    }
}

```

---

Examine this program, starting with `main()`. The call to `srand()` initializes the random number generator, which is used to simulate obstacles in the road. Then the function `StartCar()` simply prints the message Car started, letting the user know that the trip is about to begin.

The program simulates the trip with a while loop that iterates until `at_destination` becomes TRUE (1). In the loop, the car moves forward by calling the function `SteerCar()`. This function prints the message Driving... and moves the car one unit closer to the destination. When the integer `position` is equal to the destination, this function returns a 1, indicating that the trip is over. Otherwise, it returns 0.

Of course, the car's driver must always watch for obstacles. The function `FindObstacle()` acts as the driver's eyes by looking for obstacles and reporting what it finds. In this function, each time the random number generator comes up with a 0, `FindObstacle()` informs you that something is blocking the route, by returning 1 rather than 0.

If the car reaches an obstacle, the function `BrakeCar()` puts on the brakes, and the function `ReverseCar()` backs the car up. Both functions print an appropriate message. However, `ReverseCar()` also sets the car's position back one unit—unless it was already moving backward, in which case it just reverses the direction again, setting the car back in the forward direction. (The variable `forward` keeps track of the car's current direction.) The second call to `ReverseCar()` gets the car moving forward again. Finally, when the car reaches its destination, the function `TurnOffCar()` ends the program. Here is the output from a typical run of the program:

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

Car started.

Driving...

Driving...

Driving...

Driving...

Look out! There's something in the road!

Braking.

Backing up.

Driving...

Driving...

Driving...

Driving...

Driving...

Look out! There's something in the road!

Braking.

Backing up.

Driving...

Driving...

Driving...

Ah, home at last.

Turning off car.

Listing 35.2 is the object-oriented version of the program. This version includes the same functions and data. However, now everything unique to a car is encapsulated as part of the Car object.

**On the CD**

The complete source code and executable file for the Car2 application can be found in the Chap35\Car2 directory on this book's CD-ROM.

### Listing 35.2 Car2.cpp—The Object-Oriented Car Program

```

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#define HOME 10

class Car
{
    int test, position, forward;

public:
    Car(int destination);
    void StartCar(void) { cout<<"Car started." << endl; getch(); }
    int SteerCar(void);
    void BrakeCar(void) { cout<<"Braking." << endl; getch(); }
    void ReverseCar(void);
    void TurnOffCar(void) { cout<<"Turning off car." << endl; getch(); }

```

```

};

Car::Car(int destination)
{
    srand((unsigned)time(NULL));
    test = destination;
    forward = 1;
    position = 0;
}

int Car::SteerCar(void)
{
    cout << "Driving..." << endl;
    getch();
    if (++position == test) return 1;
    return 0;
}

void Car::ReverseCar(void)
{
    if (forward)
    {
        cout << "Backing up." << endl;
        getch();
        --position;
        forward = 0;
    }
    else forward = 1;
}

int FindObstacle(void)
{
    int r = rand() % 5;
    if (r) return 0;
    return 1;
}

int obstacle, at_destination = 0;
Car car(HOME);

void main()
{
    srand((unsigned)time(NULL));
    car.StartCar();
    while (!at_destination)
    {
        at_destination = car.SteerCar();
        obstacle = FindObstacle();
        if (obstacle && !at_destination)
        {
            cout << "Look out! There's something in the road!" << endl;
            getch();
            car.BrakeCar();
            car.ReverseCar();
            car.ReverseCar();
        }
    }
}

```

```
    cout << "Ah, home at last." << endl;
    car.TurnOffCar();
}
```

---

Because the program encapsulates much of the data into the class `Car`, rather than using global variables as in the first version, fewer variables are passed to functions that make up the car. This points out a subtle stylistic difference between structured programming and object-oriented programming. The first version of the program passed variables into functions—even though those variables were global—so the programmer had a clear idea about what data the function used. This is a form of self-documentation; the style of the code says something about what the code does.

In an object, the encapsulated data members are global to the object's function members, yet they are local to the object. They are not global variables. Because objects represent smaller portions of an entire program, there's no need to pass data members into member functions to help document a function's purpose. Objects are usually concise enough that this type of self-documentation is unnecessary. In Listing 35.2, no variables are passed into functions (except into the class's constructor).

Another advantage of the object-oriented approach taken in Listing 35.2 is that the `Car` object is clearly defined. All of the data and functions required for a car (at least, all that are needed for this simple computer car) are encapsulated into the class. That means there is less clutter in the main program. It also means that the code is more logically organized. In Listing 35.1, you have no clear idea of what makes up a car. The functions and data needed for the entire program are defined on the same level. For example, whereas starting a car is clearly a function of a car, finding an obstacle is not. (If you don't agree, go out to your car, climb in, and press the Find Obstacle button.) Yet the scope of both the `StartCar()` and `FindObstacle()` functions are the same. This is also true of the data. Whereas the car's destination, position, and direction all are information that help define a car, obstacles are not. You don't need an obstacle to drive a car; you do need a destination, a position, and a direction.

In Listing 35.2, every element that makes up a car is part of the class. To drive the car, the program doesn't need to deal with the car's data members. The class takes care of them. The only data that the program needs from `Car` is whether the car has arrived at its destination. The only function left in the main program is `FindObstacle()`, the one function in the program that has nothing to do with being a car. Finding obstacles is not a car's job. In all of these ways, encapsulation makes the programming task more logical and organized.

## Classes as Data Types

*Classes* are really nothing more than user-defined data types. As with any data type, you can have as many instances of the data type as you need. For example, you can have more than one car in the car program, each with its own destination. This is because a class is really nothing more than a custom data type. After you have created a data type, you can create as many instances of that data as you need.

For example, one standard data type is an integer. It's absurd to think that a program can have only one integer. You can declare many integers, just about all that you want. The same is true of classes. After you define a new class, you can create many instances of the class. Each instance (called an *object*) normally has full access to the class's member functions and gets its own copy of the data members. In the car simulation, you can create two cars, each with its own destination, as in the following:

```
Car car1(10), car2(20);
```

Although these cars derive from the same class, they are completely separate objects. The object `car2` has to go twice as far as `car1` to reach its destination.

## Header Files and Implementation Files

In Listing 35.2, all the program code is in a single file. This makes it easy to compare the first version with the second. When using object-oriented programming techniques, however, it's standard practice to place each class into two files of its own. The first, the *header file*, contains the class's definition. Usually, the header file contains all of the information that you need to use the class. The header file

traditionally has an .H extension. The actual implementation of a class's functions goes into the *implementation file*, which usually has the extension .CPP.

The header and implementation files for the Car class are shown in Listings 35.3 and 35.4, respectively. Note that the class definition has been slightly modified by adding the keyword `protected` to the data member section. This is done so that derived classes can access these data members. (I discuss inheritance in the next section.)

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY IT KNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE

[an error occurred while processing this directive]

[Previous](#)[Table of Contents](#)[Next](#)

### Listing 35.3 Car.h—The Car Class's Header File

```

#ifndef _CAR_H
#define _CAR_H

class Car
{
protected:
    int test, position, forward;

public:
    Car(int destination);
    void StartCar(void)
        { cout<<"Car started." << endl; getch(); }
    int SteerCar(void);
    void BrakeCar(void)
        { cout<<"Braking." << endl; getch(); }
    void ReverseCar(void);
    void TurnOffCar(void)
        { cout<<"Turning off car." << endl; getch(); }
};

#endif

```

### Listing 35.4 Car.cpp—The Car Class's Implementation File

```

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#include "car.h"

Car::Car(int destination)
{
    srand((unsigned)time(NULL));
    test = destination;
    forward = 1;
    position = 0;
}

int Car::SteerCar(void)
{
    cout << "Driving..." << endl;
    getch();
    if (++position == test) return 1;
}

```

Brief Full

- Advanced
- Search
- Search Tips

**BROWSE**  
BY TOPIC

```

        return 0;
    }

void Car::ReverseCar(void)
{
    if (forward)
    {
        cout << "Backing up." << endl;
        getch();
        --position;
        forward = 0;
    }
    else forward = 1;
}

```

---

## Inheritance

**Inheritance** enables you to create a class that is similar to a previously defined class but that still has some of its own properties. Consider the car simulation. Suppose you want to create a car that has a high-speed passing gear. In a traditional program, that would require a lot of code modification. As you modified the code, you would probably introduce bugs into a tested program. To avoid these hassles, use the object-oriented approach: Create a new class by inheritance. This new class inherits all of the data and function members from the base class (the class from which the new class is derived). (You can control the level of access with the public, private, and protected keywords.)

Listings 35.5 and 35.6 show the header and implementation files for a new class of car, `PassingCar`. This car inherits the member functions and data from its base class, `Car`, and adds two member functions of its own. The constructor, `PassingCar()`, does nothing but pass parameters to the base class's constructor. The member function `Pass()`, however, is unique to `PassingCar`. This is the function that gives the new car its passing gear. (Ignore the keyword `virtual` for a moment. You'll learn about virtual functions in the next section.)

If you look at Listing 35.6, you see that `Pass()` is similar to `Car`'s `SteerCar()` function, the difference being that `Pass()` increments the car's position by two units rather than one, which simulates a faster speed. Remember that although `PassingCar` has a new passing gear (implemented in the `Pass()` function), it still has access to `SteerCar()`.

### Listing 35.5 `PassingCar.h`—The `PassingCar` Class's Header File

---

```

#ifndef _PASSCAR_H
#define _PASSCAR_H

#include "car.h"

class PassingCar: public Car
{
public:
    PassingCar(int destination): Car(destination) {}
    virtual int Pass(void);
};

#endif

```

---

### Listing 35.6 `PassingCar.cpp`—The `PassingCar` Class's Implementation File

---

```

#include <iostream.h>

```

```

#include <conio.h>

#include "PassingCar.h"

int PassingCar::Pass(void)
{
    cout << "Passing..." << endl;
    getch();
    position += 2;
    if (position >= test) return 1;
    return 0;
}

```

---

Listing 35.7, a new version of the simulation's main program, gives PassingCar a test drive. When you run the program, PassingCar reaches its destination a little faster because after it backs up, it makes up time by going into passing gear. By using inheritance, this program creates a new kind of car with only a few lines of code. And the original class remains unchanged (except for the addition of the protected keyword). Impressed?



The complete source code and executable file for the Car3 application can be found in the Chap35\Car3 directory on this book's CD-ROM.

---

#### **Listing 35.7 Car3.cpp—The Main Program for Testing the PassingCar Class**

---

```

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#include "PassingCar.h"

#define HOME 10

int FindObstacle(void)
{
    int r = rand() % 5;
    if (r) return 0;
    return 1;
}

int obstacle, at_destination = 0;
PassingCar car2(HOME);

void main()
{
    srand((unsigned)time(NULL));
    car2.StartCar();
    while (!at_destination)
    {
        at_destination = car2.SteerCar();
        obstacle = FindObstacle();
        if (obstacle && !at_destination)
        {
            cout << "Look out! There's something in the road!" << endl;
            getch();
            car2.BrakeCar();
            car2.ReverseCar();
        }
    }
}

```

```
        car2.ReverseCar();
        at_destination = car2.Pass();
    }
}
cout << "Ah, home at last." << endl;
car2.TurnOffCar();
}
```

---

## Polymorphism

The last major feature of object-oriented programming is ***polymorphism***. By using polymorphism, you can create new objects that perform the same functions found in the base object but that perform one or more of these functions in a different way. For example, when the previous program used inheritance, it created a new car with a passing gear. This isn't polymorphism, because the original car didn't have a passing gear. Adding the passing gear didn't change the way an inherited function worked; it simply added a new function. Suppose, however, that you want an even faster passing gear without having to change the existing classes? You can do that easily with polymorphism.

Listings 35.8 and 35.9 show the header and implementation files for a new class, called FastCar. A FastCar is exactly like a PassingCar, except that it uses its passing gear a little differently: A FastCar moves three units forward (rather than two) when passing. To do this, the program takes an already existing member function and changes how it works relative to the derived class. This is polymorphism, in which functions work differently depending on the object type, and regardless of the type of the reference to the object. Remember that when you create a polymorphic function, you must preface its definition with the keyword `virtual`.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

### Listing 35.8 FastCar.h—The FastCar Class's Header File

```

#ifndef _FASTCAR_H
#define _FASTCAR_H

#include "PassingCar.h"

class FastCar: public PassingCar
{
public:
    FastCar(int destination):
        PassingCar(destination) {}
    virtual int Pass(void);
};

#endif
  
```

### Listing 35.9 FastCar.cpp—The FastCar Class's Implementation File

```

#include <iostream.h>
#include <conio.h>

#include "FastCar.h"

int FastCar::Pass(void)
{
    cout << "High-speed pass!" << endl;
    getch();
    position += 3;
    if (position >= test) return 1;
    return 0;
}
  
```

Look at Listing 35.10, the new main program for the car simulation. To take advantage of polymorphism, the program allocates the new FastCar dynamically—that is, it creates a pointer to the base class and then uses the new operator to create the object. Remember that you can use a pointer to a base class to access any derived classes. Note also that the base class for FastCar is not Car, but rather PassingCar, because this is the first class that declares the virtual function Pass(). If you tried to use Car as a base class, the compiler would complain, informing you that Pass() is not a member of Car. One way around this is to give Car a virtual Pass() function, too. This would make all car classes uniform with respect to a base class. (And that would probably be the best program design.)



The complete source code and executable file for the Car4 application can be found in the Chap35\Car4 directory on this book's CD-ROM.

### Listing 35.10 Car4.cpp—The New Main Program for the Car Simulation

```

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#include "FastCar.h"

#define HOME 10

int FindObstacle(void)
{
    int r = rand() % 5;
    if (r) return 0;
    return 1;
}

int obstacle, at_destination = 0;
PassingCar *car3;

void main()
{
    srand((unsigned)time(NULL));
    car3 = new FastCar(10);
    car3->StartCar();
    while (!at_destination)
    {
        at_destination = car3->SteerCar();
        obstacle = FindObstacle();
        if (obstacle && !at_destination)
        {
            cout << "Look out! There's something in the road!" << endl;
            getch();
            car3->BrakeCar();
            car3->ReverseCar();
            car3->ReverseCar();
            at_destination = car3->Pass();
        }
    }
    cout << "Ah, home at last." << endl;
    car3->TurnOffCar();
}

```

---

You must use pointers with polymorphism, because the point of polymorphism is to enable you to access different types of objects through a common pointer to a base class. You might want to do this, for example, to iterate through an array of objects. To see polymorphism work, change the line

```
car3 = new FastCar(10)
```

to

```
car3 = new PassingCar(10)
```

When you run the new version, you will be back using the slower passing gear, even though both cars use a pointer to the class `PassingCar`.

Now that you've reviewed the basics of object-oriented programming and have discovered some ways

that it makes programming easier, it's time to learn some usage and style considerations unique to the object-oriented paradigm and C++.

## Classes: From General to Specific

Starting with object-oriented programming can be a daunting experience; it's unlike other programming methods and requires adherence to a new set of principles. The process of designing a class is rarely as easy as it was with the car simulation, because classes are often based on abstractions rather than physical objects like automobiles. This makes it difficult to know what parts of a program belong in the object and which don't. Moreover, a complex program has many classes, many of which are derived from classes that might have been derived from still other classes. And each class might have many data and function members. Obviously, designing classes requires some thought and the careful application of the object-oriented philosophy.

The first step in designing a class is to determine the most general form of an object in that class. For example, suppose you're writing a graphics program and you need a class to organize the types of shapes it can draw. (In this new class, you'll draw only points and rectangles, to keep things simple.)

Determining the most general class means determining what the objects in the class have in common. Two things that come to mind are color and position. These attributes become data members in the base class. Now, what functions must a shape perform? Each shape object needs a constructor and a way to draw itself on-screen. Because drawing a point is different from drawing a square, you'll need to put polymorphism to work and use a virtual function for the drawing task.

Listing 35.11 is the header file for a Shape class. This class needs no implementation file because the class is fully implemented in the header file. The constructor is implemented inline, and the pure virtual function DrawShape() requires no implementation because it is only a placeholder for derived classes.

---

### Note:

A pure virtual function not only requires no implementation, but also by definition cannot have an implementation. The derived class must provide the implementation.

---

---

### Listing 35.11 Shape.h—The Header File for the Shape Class

---

```
#ifndef _SHAPE_H
#define _SHAPE_H

class Shape
{
protected:
    int color, sx, sy;

public:
    Shape(int x, int y, int c)
        { sx=x; sy=y; color=c; }
    virtual void DrawShape(void) = 0;
};

#endif
```

---

As you can see from Listing 35.11, Shape does nothing but initialize the data members color, sx, and sy, which are the color and X,Y coordinates of the object. To do anything meaningful with the class, you must derive a new class for each shape that you want to draw. Start with the point. Listings 35.12 and 35.13 are the header and implementation files for this new class.

---

### Listing 35.12 Point.h—The Header File for the Point Class

---

```
#ifndef _POINT_H
#define _POINT_H

#include "shape.h"

class Point: public Shape
{
public:
    Point(int x, int y, int c): Shape(x, y, c) {};
    virtual void DrawShape(void);
};

#endif
```

---

### **Listing 35.13 Point.cpp—The Point Class’s Implementation File**

---

```
#include "point.h"

void Point::DrawShape(void)
{
    putpixel(sx, sy, color);
}
```

---

The constructor for this class does nothing but pass parameters to the base class’s constructor; thus, it is implemented inline. The DrawShape() function, however, must draw the shape—in this case, a dot on-screen at the coordinates and in the color found in the sx, sy, and color data members. This function, too, is short and could have been implemented inline. However, to keep the program construction parallel with the next example, there is a separate implementation file for the Point class.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb’s [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

---

#### Note:

Remember: To keep the program listings as simple as possible, these are not Windows programs. The Windows API has no putpixel() function.

---

Listing 35.14 is the test program for the shape classes. Because polymorphism is used to create shape classes and because each class is derived from the Shape base class, the program can test a new shape class simply by changing the type of object created by the new operator. If you were to run the program, a dot would appear in the middle of your screen.

#### Listing 35.14 TestShape.cpp—The Test Program for the Shape Classes

---

```
#include "point.h"
//#include "rectngle.h"
//#include "barrec.h"

void main()
{
    Shape* r;
    r = new Point(100, 100, WHITE);
    r->DrawShape();
    delete r;
}
```

---

To make things interesting, add a second shape, Rectngle, to the classes. Rectngle is also derived from Shape. Listings 35.15 and 35.16 show the files for this new class.

#### Listing 35.15 Rectngle.h—The Header File for the Rectngle Class

---

```
#ifndef _RECTNGLE_H
#define _RECTNGLE_H

#include "shape.h"

class Rectngle: public Shape
{
protected:
    int x2, y2;
```

```
public:
    Rectngle(int x1, int y1, int w, int h, int c);
    virtual void DrawShape(void);
};

#endif
```

---

### **Listing 35.16 Rectngle.cpp—The Implementation File for the Rectngle Class**

---

```
#include "rectngle.h"

Rectngle::Rectngle(int x1, int y1, int w, int h, int c):
    Shape(x1, y1, c)
{
    x2 = sx + w;
    y2 = sy + h;
}

void Rectngle::DrawShape(void)
{
    setcolor(color);
    rectangle(sx, sy, x2, y2);
}
```

---

If you want to test this new class, in the main program, change the line

```
r = new Point(100, 100, WHITE);
```

to something like

```
r = new Rectngle(200, 200, 100, 100, WHITE);
```

Thanks to polymorphism, this is the only change (outside of uncommenting the line `#include "rectngle.h"`) that you need in the main program to draw a rectangle.

The class `Rectngle` is more complicated than the `Point` class. To draw a rectangle, the program needs—in addition to the rectangle's X,Y coordinates—the rectangle's width and height. This means that `Rectngle`'s constructor does more than send parameters to the base class. It also initializes two extra data members, `x2` and `y2`. `Rectngle`'s `DrawShape()` function, too, is more complicated than `Point`'s, because drawing a rectangle takes more work than drawing a dot.

So far, you've gone from an abstract shape, which did nothing but initialize a couple of data members, to drawing two simple shapes on-screen. You can now move down another level, from the general shape of a rectangle to a more specific type: a rectangle with a colored bar at the top. This type of rectangle might, for example, be the starting point for a labeled window. Listings 35.17 and 35.18 are the source code for the `BarRec` class.

### Listing 35.17 BarRec.h—The Header File for the BarRec Class

---

```
#ifndef _BARREC_H
#define _BARREC_H

#include "rectngle.h"

class BarRec: public Rectngle
{
public:
    BarRec(int x1, int y1, int w, int h, int c):
        Rectngle(x1, y1, w, h, c) {}
    virtual void DrawShape(void);
};

#endif
```

---

### Listing 35.18 BarRec.cpp—The Implementation File for the BarRec Class

---

```
#include "barrec.h"

void BarRec::DrawShape(void)
{
    setcolor(color);
    rectangle(sx, sy, x2, y2);
    setfillstyle(SOLID_FILL, RED);
    bar(sx+2, sy+2, x2+-2, sy+15);
}
```

---

If this were a real program, you could test the new shape by changing the new statement in the main program to

```
r = new BarRec(200, 200, 100, 100, WHITE);
```

Then, when you ran the program, the new type of rectangle object would appear on-screen.

You could easily continue creating new types of rectangles. For example, if you want a rectangle with both a bar at the top and a double-line border, you can derive a new type from BarRec, overriding its virtual DrawShape() with one of its own. (This new function would probably need to call its base's DrawShape() function to draw the bar at the top and then do the extra drawing required for the double border.)

---

#### Note:

The need to call a base class's version of an overridden function should be a familiar concept to you as an MFC programmer. Often, when you override member functions of an MFC class, you not only provide your own specialized code, but also call the base class's version of the function to ensure that the

function performs all of the tasks it was designed to perform for the class.

---

---

#### **NOTE:**

By using the general-to-specific method of creating classes, you end up with extremely flexible code. You'll have many classes from which to choose when it comes time to derive a new one. Moreover, classes will be less complex than they would be if you tried to cram a lot of extra functionality into them. Remember that the more general you make your classes, the more flexible they are.

---

## **Single-Instance Classes**

Object-oriented programming means power. When programmers first experience this power, they find it irresistible. Suddenly, they're using objects for everything in their programs, without thinking about whether each use is appropriate. Remember that C++ is both an object-oriented language and a procedural language. In other words, C++ programmers get the best of both worlds and can develop a strategy for a particular programming problem that best suits the current task. That strategy might or might not include an object-oriented approach.

Classes are most powerful when used as the basis for many instances. For example, soon you'll delve more deeply into object-oriented programming techniques by putting together a string class (in the section "Developing a String Class"). After developing the class, you're likely to have many instances of strings in your programs, each inheriting all the functionality of its class.

Nothing comes free, however. There is always a price. For example, to call an object's member functions, you must use a more complicated syntax than you need for ordinary function calls; you must supply the object and function name. Moreover, creating classes is a lot of work. Why go through all of the extra effort if the advantages don't outweigh the disadvantages?

Although classes are most appropriate when used to define a set of objects, there are times when creating a single-instance class is a reasonable strategy. For example, many DOS programmers created classes to handle the mouse. Although you'll never have more than one mouse operating simultaneously, writing mouse functions into a single-instance class enables a programmer to conveniently package and organize routines that he or she will need often.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Brief Full

- [Advanced Search](#)
- [Search Tips](#)

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

Generally, a single-instance class is okay for wrapping up a big idea, like a screen display, a mouse driver, or a graphics library. It might not, however, be appropriate for smaller uses that would suffer from the overhead inherent in using classes. Remember that although you're programming in C++, you still can use simpler data structures like structures and arrays. When you need to create a new data type, don't automatically assume that the object-oriented approach is best. Often, it's not.

## Responsible Overloading

One of the things that differentiates C from C++ is function and operator overloading. **Overloading** is the capability to create several versions of a function or operator, each version of which has an identical name but which requires different arguments. For example, in C++, you can have two functions named Sum(), one that adds integers and another that adds floating-point numbers. When you call Sum() in a program, the compiler can tell which function you mean by checking the function's parameters.

The capability of C++ to overload functions and operators offers immense flexibility. You no longer have to come up with different names for functions that, although they take different parameters, perform virtually identical operations. You can simply write several versions of the function, using the same name, each version with its own set of arguments. As you've already learned, however, powerful techniques are often misused. In this section, you'll examine function- and operator-overloading etiquette.

### Overloading versus Default Arguments

There's no question that function overloading is a great feature of C++ programming. However, when overused, it can make code more difficult to understand. If nothing else, having several versions of a function considerably increases program maintenance. The solution? Default arguments also enable you to call functions with different parameters, but without resorting to overloading. For example, consider the following overloaded function:

```
int Example(int x);  
int Example(int x, int y);
```

Because of overloading, you can call the function Example() with one or two integer arguments:

```
Example(1);
```

```
Example(1,2);
```

This adds much to the function's flexibility. However, do you really need two copies of the function to get this flexibility? Not really. By using default arguments, you can create one version of `Example()` that accepts either one or two integer arguments:

```
int Example(int x, int y = 0);
```

This new function retains the flexibility of the overloaded function, but without the extra baggage. Of course, you can't always replace overloaded functions with default arguments. For example, if the parameter types of overloaded functions are different, the default-argument technique won't work. The following overloaded function cannot be written using default arguments:

```
int Example(int x);  
float Example(float x);
```

You can't have a default type, only a default value. When you get the urge to overload a function, first consider whether it would be more expedient to use default arguments.

## Using Operator Overloading Logically

You've seen how function overloading can be both bounty and bane. Operator overloading, too, requires thought before you use it. Although the use of default arguments doesn't apply here, there are still important considerations. The most important is using overloaded operators logically—in other words, using them as they were originally designed to be used.

Using operator overloading, you can make any of C++'s operators perform whatever task you want. For example, the `+` operator sums two values. Without operator overloading, this operator can be used only on C++'s built-in data types—in other words, types like `int`, `float`, and `long`. Suppose, however, that you want to add two arrays and assign the result to a third array. You can then overload the `+` and `=` operators in an array class so that they can take arrays as arguments. Assuming you've done this, what do you suppose the following line would do (where `a`, `b`, and `c` are objects of your array class)?

```
c = a + b;
```

You'd expect that the equal sign acts as an assignment operator because that is normally its purpose. Similarly, you'd expect that the `+` operator summed the elements of each array. (You can find the code that performs this overloading in Listing 35.19.) What you wouldn't expect is for the sum operator to take, for example, two two-element arrays and combine them into a four-element array. This type of operation would not be consistent with the operator's conventional usage.



The complete source code and executable file for the Array application can be found in the `Chap35\Array` directory on this book's CD-ROM.

### Listing 35.19 Array.cpp—Defining Array Operators

---

```
#include <iostream.h>
#include <conio.h>

class Array
{
    int a[2];

public:
    Array(int x=0, int y=0);
    void Print(void);
    Array operator=(Array b);
    Array operator+(Array b);
};

Array::Array(int x, int y)
{
    a[0] = x;
    a[1] = y;
}

void Array::Print(void)
{
    cout << a[0] << ' ' << a[1] << endl;
}

Array Array::operator=(Array b)
{
    a[0] = b.a[0];
    a[1] = b.a[1];
    return *this;
}

Array Array::operator+(Array b)
{
    Array c;

    c.a[0] = a[0] + b.a[0];
    c.a[1] = a[1] + b.a[1];
    return c;
}

void main()
{
    Array a(10, 15);
    Array b(20, 30);
    Array c;
```

```
a.Print();  
b.Print();  
c.Print();  
c = a + b;  
c.Print();  
getch();  
}
```

---

Operators should perform as expected. This means more than simply using them for the expected operation. It also means that they should perform that operation in a way that is consistent with the language's implementation. For example, look at the code for the + operator in the array class (refer to Listing 35.19). Notice that the source arrays are unchanged by the operation. Instead, a third array is used to hold the results of the addition. This third array is returned from the function. This is how you expect the addition operator to work in C++. Contrast this with the way an addition instruction works in assembly language, by storing the result of the operation into one of the two operands. In most assembly languages, one of the operands is changed by the operation. In C++, it is not.

---

**Note:**

Overloading functions and operators is a powerful technique. Like all powerful features of a language, however, this one must be used with thought and style. Don't use overloading when a simpler method will do, and ensure that overloaded operators perform in the expected way.

---

## When to Use Virtual Functions

Using virtual functions, you can create classes that, like the simple graphics demonstration in a previous section ("Classes: From General to Specific"), perform the same general functions but perform those functions differently for each derived class. Like overloading, however, virtual functions are often misused.

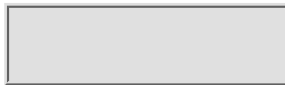
<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief    Full  
 • [Advanced Search](#)  
 • [Search Tips](#)



[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

Before using a virtual function, consider how the classes in the hierarchy differ. Do they need to perform different actions? Or do the classes require only different values? For example, in the shapes demonstration, the program used virtual functions so that each class could draw its shape properly. Every shape object must know how to draw itself; however, every object needs to do it differently. Drawing a shape is an action. It's inappropriate, though, to use a virtual function to assign a color to an object. Although each shape object has its own color attribute, the color attribute is a value rather than an action, and so it is best represented by a data member in the base class. Using polymorphism to set an object's color is like trying to kill a mosquito with a machine gun.

---

**TIP:**

Make sure that when you use virtual functions you are creating classes that differ in action rather than value.

---

## Developing a String Class

In the preceding sections, you reviewed object-oriented program design and some C++ style considerations. In this section, you will apply much of what you learned before to create a string class for your C++ programs. (Yes, MFC provides the CString class, which you can use to do your string handling. However, designing a similar string class enables you to create a class that better suits your needs and tastes—not to mention that it's a great exercise in class design.)

Handling strings in C has always been tougher than pulling meat from a lion's mouth, especially when compared with the excellent string-handling capabilities of many other high-level languages. Unfortunately, in C, you can't create classes and overload operators, so good string handling can't be incorporated into the language, even by user-written routines. For example, in C, strings cannot be assigned by the simple expression  $A = B$ .

Thankfully, you're a C++ programmer. By using C++'s overloading capabilities—both for functions and operators—as well as taking advantage of its object-oriented programming features, you can create a string class that provides all the string-manipulation features of languages like Pascal.

## Choosing a Storage Method

To design your own string class, ask yourself a couple of questions. First, how

should you represent the string? There are two approaches you can take: a standard character array or dynamically allocated memory. Both approaches have strong and weak points. For example, the character-array approach is the simplest, enabling you to use C++'s array-handling capabilities without having to worry about the details of memory allocation.

On the other hand, using a character array is the least flexible of the choices, because you must choose a maximum string size and stick with it. Moreover, your character array will take up the same amount of memory regardless of the actual length of the string. For example, suppose you choose an 81-element character array, which has enough space for 80 characters plus a null. Then each string that you create will take up 81 bytes of memory, although the string data might be only a few bytes.

By dynamically allocating space for a string and by grabbing only the memory that you actually need to contain the string, you can use memory more efficiently. This method, however, requires a lot of program overhead. You must write code to handle memory allocation and deallocation, check for allocation errors and null strings, keep track of a string's size, and take care of other messy details. In fact, the extra code required for a dynamically allocated string class would probably use as much memory as you'd waste with the character-array approach (depending on the number of strings a program uses, of course). To keep things simple and clean, then, the class presented in this chapter uses the character-array method, with an 81-element array.

## **Determining the Class's Functionality**

Now that you've chosen a type of storage, consider how your programs will use strings. To be as flexible as possible, your programs must be able to handle two types of strings: standard character arrays and String objects (instances of the String class). For example, you must allow string assignments such as `str1 = str2` (in which `str1` and `str2` are String objects) and `str1 = "STRING"` (in which `str1` is a String object and "STRING" is a standard C character array). This means that you're going to have to overload functions to accept either type of parameter.

You now have a general strategy for string storage and string usage. Next you need to decide what functions will give you the string-handling power that you want. The basic functions required in a string class vary with the needs of each programmer; everyone programs differently. Moreover, each program has its own requirements. The string class in this chapter contains the most-often-used functions. When the string class is complete and you've used it in your own programs, you might find that you need additional functions. No problem! Add them by modifying the original class. When you understand how the basic class was created, you should have no difficulty modifying it to meet specific needs.

What are the basic functions that a string class requires? You can answer this question easily by examining a popular, high-level language with good string handling, such as Pascal. Examining Borland's Turbo Pascal yields a list of important string-handling functions. These functions are listed here:

- String construction and destruction

- String assignment
- String concatenation
- String comparison
- String searches
- String insertion
- String deletion
- String extraction
- String retrieval

Each function in the string class will be covered in its own section. Before you get started, however, look at Listing 35.20, the header file for the String class. Compare it with Table 35.1, which lists each function and its usage.

Obviously, if you understand how to use the class, you'll better understand the programming involved. You might also want to look over Listing 35.34, near the end of this chapter, to get a general idea of how the string functions are used in a program.

#### **Listing 35.20 STRNG.H—The Header File for the String Class**

---

```
#ifndef _STRNG_H
#define _STRNG_H

#include <string.h>
#include <conio.h>
#include <iostream.h>

class String
{
    char s[81];

public:
    String(char *ch);
    String(String &amp;str) { strcpy(s, str.s); }
    void GetStr(char *ch, int size);
    String GetSubStr(unsigned index, int count);
    void Delete(unsigned index, int count);
    void Insert(String str, int index);
    void Insert(char *ch, int index);
    int Length() { return strlen(s); }
    int Pos(String str);
    int Pos(char *ch);
    String operator=(String str);
    String operator=(char *ch);
    String operator+(String str);
    String operator+(char *ch);
    int operator==(String str);
    int operator==(char *ch);
    int operator!=(String str);
```

```
int operator!=(char *ch);
int operator<(String str);
int operator<(char *ch);
int operator>(String str);
int operator>(char *ch);
int operator>=(String str);
int operator>=(char *ch);
int operator<=(String str);
int operator<=(char *ch);
};

#endif
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**Table 35.1 String Class Description**

String Function	Description
Add()	Adds an image to the image list.
Attach()	Attaches an existing image list to an object of the CImageList class.
BeginDrag()	Starts an image-dragging operation.
Create()	Creates an image list control.
DeleteImageList()	Deletes an image list.
Detach()	Detaches an image list from an object of the CImageList class.
DragEnter()	Locks a window for updates and shows the drag image.
String(String &amp;str)	These are the class's constructors. The constructor accepts as a parameter either a character array or a String object.
void GetStr(char *ch, int size)	This function retrieves a String and places it into a character array. The parameter ch is a pointer to the destination character array, and size is the length of the destination array.
String GetSubStr(unsigned index, int count)	This function returns a String made up of count characters. The characters are extracted from the String starting with the character at position index.
void Delete(unsigned index, int count)	This function deletes count characters from the String object, starting with the character at position index.
void Insert(String str, int index)	This function inserts str into a String, at position index.
void Insert(char *ch, int index)	This function inserts a character array pointed to by ch into a String, starting at string character position index.
int Length()	This function returns the length of a String.
int Pos(String str)	This function returns the character position of the first occurrence of str within a String.
int Pos(char *ch)	This function returns the character position of the first occurrence of ch (a character array) within a String.

<code>String operator=(String str)</code>	Assigns <code>str</code> to a <code>String</code> .
<code>String operator=(char *ch)</code>	Assigns <code>ch</code> (a character array) to a <code>String</code> .
<code>String operator+(String str)</code>	Concatenates a <code>String</code> and <code>str</code> .
<code>String operator+(char *ch)</code>	Concatenates a <code>String</code> and <code>ch</code> (a character array).
<code>int operator==(String str)</code>	Compares a <code>String</code> with <code>str</code> , returning 1 if they are equal or 0 if they are not equal.
<code>int operator==(char *ch)</code>	Compares a <code>String</code> to <code>ch</code> (a character array), returning 1 if they are equal or 0 if they are not equal.
<code>int operator&lt;(String str)</code>	Returns 1 if <code>String</code> is less than <code>str</code> , or else returns 0.
<code>int operator&lt;(char *ch)</code>	Returns 1 if <code>String</code> is less than <code>ch</code> , or else returns 0.
<code>int operator&gt;(String str)</code>	Returns 1 if <code>String</code> is greater than <code>str</code> , or else returns 0.
<code>int operator&gt;(char *ch)</code>	Returns 1 if <code>String</code> is greater than <code>ch</code> , or else returns 0.
<code>int operator&lt;=(String str)</code>	Returns 1 if <code>String</code> is less than or equal to <code>str</code> , or else returns 0.
<code>int operator&lt;=(char *ch)</code>	Returns 1 if <code>String</code> is less than or equal to <code>ch</code> , or else returns 0.
<code>int operator&gt;=(String str)</code>	Returns 1 if <code>String</code> is greater than or equal to <code>str</code> , or else returns 0.
<code>int operator&gt;=(char *ch)</code>	Returns 1 if <code>String</code> is greater than or equal to <code>ch</code> , or else returns 0.

---

## String Construction and Destruction

Thanks to object-oriented programming, string initialization can be handled by the class's constructor. This means that you can create and initialize a new `String` with a single declaration—for example, `String str("TEST STRING")`—or you can create an empty string; for example, `String str("")`.

By using conventional character arrays, rather than dynamically allocated memory, the class needs no string destructor. The class creates nothing that can't be handled automatically by C++. If, however, the string class used dynamically allocated memory, its destructor would have been responsible for releasing memory allocated to a string.

Finally, as mentioned previously, the `String` class must deal with both `String` objects and standard C character arrays. Therefore, it needs to overload the constructor. One version constructs a `String` from an existing `String` and another constructs a `String` from a character array. The former is implemented inline:

```
String(String &amp;str) { strcpy(s, str.s); }
```

The constructor doesn't need to worry about the length of `str.s`. It's already a `String` object; thus, it is guaranteed to be 80 characters or less. To construct the new `String`, the function simply copies one string

into the other.

Listing 35.21 is the source code for the character array version of the constructor.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------



Brief

Full

Advanced

Search

Search Tips



[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

### Listing 35.21 lst35\_21.cpp—The String Class’s Constructor for Character Arrays

```
String::String(char *ch)
{
    strncpy(s, ch, 80);
    s[80] = 0;
}
```

Here, the constructor uses `strncpy()` to ensure that no more than 80 characters will be copied from the source array to the string object’s storage. The constructor then places a null in `ch[80]`, because `strncpy()` isn’t guaranteed to place the null.

### String Assignments

To conveniently handle string assignments, the string class overloads C++’s assignment operator (`=`). In fact, it overloads it twice: once for character arrays and once for String objects. An assignment operator would be crippled if it couldn’t accept string constants, which are represented in C++ by character arrays. Listing 35.22 is the source code for the String version of the `=` operator.

### Listing 35.22 lst35\_22.cpp—The Assignment Operator for String Objects

```
String String::operator=(String str)
{
    strcpy(s, str.s);
    return *this;
}
```

As with the string constructor, the source String is already in the acceptable format, so the function can just copy it directly into the destination String. Note the use of the pointer `this`, which is a pointer to the object that called the function. Every call to a class’s function gets `this` as a hidden parameter. So the function in Listing 35.22 returns a pointer to the object. This makes it possible to use the new assignment operator in such expressions as `str1 = str2 = “TEST STRING”`. Also, this is the way that programmers expect the C++ assignment operator to work. You should avoid giving programmers nasty surprises. Surprises make them cranky. Listing 35.23 is the character array version of the

function.

### **Listing 35.23 lst35\_23.cpp—The Assignment Operator for a Character Array**

---

```
String String::operator=(char *ch)
{
    strncpy(s, ch, 80);
    s[80] = 0;
    return *this;
}
```

---

This version works much like the first, except that the function can no longer assume that the source character array is 80 characters or less. As with the character array version of the constructor, therefore, the function checks the length of *ch* and truncates it if necessary. Then it uses *strcpy()* to copy the array into *s*.

## **String Concatenation**

There probably aren't too many string-intensive programs that couldn't benefit from a string-concatenation function. For example, a program might need to combine a person's first and last names, build a complete path name out of directory and file name strings, or assemble phrases into sentences.

Concatenating strings is trickier than making simple string assignments. First you must be sure that the final *String* is no longer than the allowable 80 characters. Also, as discussed previously ( in the section "Using Operator Overloading Logically"), you must use the *+* operator in the expected way. Specifically, you must not change either of the source strings, but rather return a third string that is the concatenation of the source strings. And you need two versions of the function, one for *Strings* and one for character arrays. Listing 35.24 is the *String* version.

### **Listing 35.24 lst35\_24.cpp—The Concatenation Operator for *String* Objects**

---

```
String String::operator+(String str)
{
    char ch[161];
    String str1("");

    strcpy(ch, s);
    strcpy(&ch[strlen(s)], str.s);
    ch[80] = 0;
    strcpy(str1.s, ch);
    return str1;
}
```

---

Although the function concatenates two strings, you might wonder why only one string is listed in the function's parameters. This is because the other source string is the String object that called the function. Which object calls the operator function? With operators, the object on the left is always the one that makes the function call. For example, in the statement `str2 = str1 + "TEST STRING"`, the object `str1` calls the concatenation function. You don't need to pass `str1` as a parameter, because you already have access to it from within the class.

The function in Listing 35.24 uses a 161-element character array as temporary storage for the strings being concatenated. By using this double-sized character array, the function can concatenate the two Strings (which are 80 characters or less) without worrying about overrunning the destination array. To return a String in the proper format, the function simply places a null in `ch[80]`, which truncates `ch` if it's larger than 80 characters. After concatenating the Strings, the function copies the resulting character string into a new String object, `str1`, which is the String returned.

The character array version of the concatenation function is much simpler than the String version, as shown in Listing 35.25.

---

**Listing 35.25** `lst35_25.cpp`—The Concatenation Operator for a Character Array

---

```
String String::operator+(char *ch)
{
    String str(ch);
    return *this + str;
}
```

---

Rather than duplicate a lot of code, it's much easier to convert the character array to a String object and then use the String version of the concatenation function to do the dirty work. Notice that the function uses a dereferenced `this` pointer to access the String object that called the function.

## String Comparison

Comparing strings is a particularly handy function. Often, for example, in an interactive program, you need to check a user's input against some expected response. C++ already provides string-comparison functions, but those functions can be improved by hiding their somewhat clumsy implementation inside of the String class. By overloading C++'s `==` operator, you can compare strings in a more natural way. Listing 35.26 is the implementation for both versions of this function.

---

**Listing 35.26** `lst35_26.cpp`—The Comparison Operator

---

```
int String::operator==(String str)
{
    if (strcmp(s, str.s) == 0) return 1;
}
```

```
        return 0;
    }

    int String::operator==(char *ch)
    {
        if (strcmp(s, ch) == 0) return 1;
        return 0;
    }
```

---

These functions differ only in the type of parameter that they accept. Both use the C++ function `strcmp()` to compare two character arrays. Unlike the `strcmp()` function, however, which returns a false (0) value when the strings match, the string class's comparison function returns true (1) for a match and false (0) otherwise. This is the way you would expect the `==` comparison operator to work.

The string class also includes overloaded functions for all other types of comparisons, as shown in Table 35.1. Note that the comparison functions provided there are case-sensitive. You might want to develop comparison operators that are not case-sensitive.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#)
[Table of Contents](#)
[Next](#)

## String Searches

Sometimes you might want to locate a series of characters within a string. Again, as with string comparisons, C++'s string library already provides a function for locating substrings. The `strncmp()` function works like `strcmp()`, except that it limits its comparison to the number of characters specified in the last parameter. You can easily use `strncmp()` to locate a substring and return its position. The string class's `Pos()` function uses `strncmp()` for just this task, as shown in Listing 35.27.

### Listing 35.27 `lst35_27.cpp`—The `Pos()` Function

```
int String::Pos(String str)
{
    int found = 0;

    if ((str == "") || (str.Length() > Length()))
        return 0;
    int i = 0;
    while ((!found) && (i < Length()))
    {
        if (strncmp(&amps[i], str.s, str.Length()) == 0)
            found = 1;
        else ++i;
    }
    if (found) return i+1;
    return 0;
}
```

Here, the function first checks whether the passed `String` is null or is longer than the `String` that called the function. In either case, there can't possibly be a match; therefore, the function returns a 0. If the function gets past this first check, it enters a while loop that uses the index `i` to cycle through each character of the `String` object. In the call to `strncmp()`, the index is used to calculate the address of the character with which to start the compare (`&amps[i]`) with the search `String`. If `strncmp()` finds a match, the flag `found` is set, which causes the loop to end. Then the value `i+1`—the position of the character that begins the substring—is returned from the function.

The character array version of `Pos()`, like the character array version of the concatenation function, simply converts the character array to a `String` object and then calls the `String` version of `Pos()`. This trick makes adding a character array version of most functions



easier than toasting marshmallows in a forest fire. Listing 35.28 is the character array version of the function.

---

**Listing 35.28 lst35\_28.cpp—The Character Array Version of the Pos() Function**

---

```
int String::Pos(char *ch)
{
    String str(ch);
    return Pos(str);
}
```

---

## String Insertion

Another handy string operation—one that’s similar to string concatenation—is *string insertion* (placing one string into another). The String class accomplishes this task with the function Insert(). The String version is shown in Listing 35.29.

---

**Listing 35.29 lst35\_29.cpp—The String Version of Insert()**

---

```
void String::Insert(String str, int index)
{
    char ch[161];

    if ((index <= Length()) && (index > 0))
    {
        strncpy(ch, s, index-1);
        strcpy(&ch[index-1], str.s);
        strcpy(&ch[strlen(ch)], &s[index-1]);
        ch[80] = 0;
        strcpy(s, ch);
    }
}
```

---

This function first checks for a valid index. If the index is okay, it uses strncpy() to copy all of the characters, up to the index, into a temporary character array. Then it adds the string that you want to insert to the array. Finally, it copies the remaining characters in the original String into the temporary array, placing a null in ch[80] to ensure that the returned string is 80 characters or less, as required by the String class. Note that this function returns no value; it operates directly on the String object that calls the function.

The character array version of Insert(), again, does nothing more than convert the array to a String object and then call the String version of the function. Listing 35.30 is that version of the function.

---

**Listing 35.30 lst35\_30.cpp—The Character Array Version of Insert()**

---

```
void String::Insert(char *ch, int index)
{
```

```
String s1(ch);  
Insert(s1, index);  
}
```

---

## String Deletion

The opposite of insertion is, of course, deletion. A *string deletion* function enables you to remove a substring from a String object. In the String class, the function Delete() does the job, as shown in Listing 35.31.

### Listing 35.31 lst35\_31.cpp—The Delete() Function

---

```
void String::Delete(unsigned index, int count)  
{  
    String s1("");  
  
    if ((index <= strlen(s)) && (index > 0) && (count > 0))  
    {  
        strncpy(s1.s, s, index-1);  
        if ((index+count-1) <= strlen(s))  
            strcpy(&s1.s[index-1], &s[index+count-1]);  
        else s1.s[index-1] = 0;  
        *this = s1;  
    }  
}
```

---

This function works similarly to the insertion function. It first checks that the index is valid. It also checks that count is greater than 0. If the index or the count is invalid, the function does nothing. If the index is okay (greater than 0 and less than or equal to the length of the string) and count is greater than 0, index-1 characters are copied from the beginning of the source String (the one that called the function) into a temporary String. Then the function checks whether the source String, starting at index, contains at least count characters. If it does, the characters starting at index+count-1 are added to the temporary String. Otherwise, if count is larger than the number of remaining characters in the source String, the function just adds a null to the temporary String, which effectively deletes all remaining characters in the String. Note that in the last line, \*this = s1, the assignment operator is the one defined for the String class, not the usual C++ assignment operator.

## String Extraction

A *string extraction* function is much like a string deletion function, except that the extraction function returns a new string containing the requested characters without deleting the characters from the original string. In the String class, the function GetSubStr() takes on this chore. Because the function takes only integer parameters, only one version is needed, as shown in Listing 35.32.

### Listing 35.32 lst35\_32.cpp—The GetSubStr() Function

---

```
String String::GetSubStr(unsigned index, int count)
{
    String s1("");

    if ((index <= strlen(s)) && (index > 0) && (count > 0))
        int c = Length() - index + 1;
        if (count > c) count = c;
        strncpy(s1.s, &amps[index-1], count);
        s1.s[count] = 0;
    }
    return s1;
}
```

---

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH  
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE  
BY TOPIC

[an error occurred while processing this directive]

[Previous](#) [Table of Contents](#) [Next](#)

As always, the function first checks for a valid index and count. If the index is valid (less than or equal to the length of the string and greater than zero) or count is less than 1, it returns a null String from the function. If the index and count are okay, the number of characters in the String starting at index are calculated, and count is adjusted if necessary. (You don't want to try to copy more characters than exist in the String.) Finally, `strncpy()` copies the requested characters into the new String object, and that String is returned from the function.

## String Retrieval

The last function in the String class enables you to convert the contents of a String back to a character array. You might need to do this, for example, to manipulate the string in a manner that is not supported by the String class. The `GetStr()` function, shown in Listing 35.33, handles the conversion task.

### Listing 35.33 `lst35_33.cpp`—The `GetStr()` Function

```
void String::GetStr(char *ch, int size)
{
    strncpy(ch, s, size-1);
    ch[size-1] = 0;
}
```

Here, the function simply copies the String object's character array into the array pointed to by `ch`. Note that it's imperative that the `size` parameter, which tells the function the size of `ch`, be correct. To be sure of this, you should always use `sizeof()` as the second parameter in a call to this function, as in the following:

```
str.GetStr(ch, sizeof(ch));
```

Why can't you use the `sizeof()` function inside `GetStr()` and avoid having to pass it off as a parameter? Because all `GetStr()` knows about `ch` is that it's a pointer to `char`; the size of a pointer is four bytes. De-referencing the pointer won't work either, because then you'd be asking for the size of the data to which `ch` pointed. What does `ch` point to? Characters, of course, which are actually integers. `GetStr()` has no way of knowing that `ch` actually points to an array of characters.

## Testing the *String* Class



That's it! You now know how to design and write a class. Listing 35.34 is the complete source code for the String class's implementation. Listings 35.35 through 35.37 make up a Windows program that tests the new class and show how each function is called. You can find the source code, as well as the executable file, for this program in the `Chap35\strg` folder on this book's CD-ROM.

**Listing 35.34 STRGAPP.H—Header File for the CStrgApp Class**

---

```
////////////////////////////////////
// STRGAPP.H: Header file for the CStrgApp class, which
//           represents the application object.
////////////////////////////////////

class CStrgApp : public CWinApp
{
public:
    CStrgApp();

    // Virtual function overrides.
    BOOL InitInstance();
};
```

---

**Listing 35.35 STRGAPP.CPP—Implementation File for the CStrgApp Class**

---

```
////////////////////////////////////
// STRGAPP.CPP: Implementation file for the CStrgApp,
//           class, which represents the application
//           object.
////////////////////////////////////

#include <afxwin.h>
#include "strgapp.h"
#include "mainfrm.h"

// Global application object.
CStrgApp StrgApp;

////////////////////////////////////
// Construction/Destruction.
////////////////////////////////////
CStrgApp::CStrgApp()
{
}

////////////////////////////////////
// Overrides
////////////////////////////////////
BOOL CStrgApp::InitInstance()
{
    m_pMainWnd = new CMainFrame();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}
```

---

**Listing 35.36 MAINFRM.H—Header File for the CMainFrame Class**

---

```

////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//             represents the application's main window.
////////////////////////////////////

class CMainFrame : public CFrameWnd
{
// Constructor and destructor.
public:
    CMainFrame();
    ~CMainFrame();

// Overrides.
protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Message map functions.
public:
    afx_msg void OnPaint();

// Protected member functions.
protected:
    void ShowStrings(CPaintDC* paintDC);

    DECLARE_MESSAGE_MAP()
};

```

---

### **Listing 35.37 MAINFRM.CPP—Implementation of the CMainFrame Class**

---

```

////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//             class, which represents the application's
//             main window.
////////////////////////////////////

#include <afxwin.h>
#include "mainfrm.h"
#include "strng.h"

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()

////////////////////////////////////
// CMainFrame: Construction and destruction.
////////////////////////////////////
CMainFrame::CMainFrame()
{
    Create(NULL, "String App", WS_OVERLAPPED | WS_SYSMENU);
}

```

```

CMainFrame::~CMainFrame()
{
}

////////////////////////////////////
// Overrides.
////////////////////////////////////
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Set size of the main window.
    cs.cx = 240;
    cs.cy = 360;

    // Call the base class's version.
    BOOL returnCode = CFrameWnd::PreCreateWindow(cs);

    return returnCode;
}

////////////////////////////////////
// Message map functions.
////////////////////////////////////
void CMainFrame::OnPaint()
{
    CPaintDC* paintDC = new CPaintDC(this);
    ShowStrings(paintDC);
    delete paintDC;
}

////////////////////////////////////
// Protected member functions.
////////////////////////////////////
void CMainFrame::ShowStrings(CPaintDC* paintDC)
{
    String s1("THE HAT");
    String s2(s1);
    String s3("");
    char ch[81];
    TEXTMETRIC tm;

    paintDC->GetTextMetrics(&tm);
    UINT position = tm.tmHeight;

    s1.GetStr(ch, sizeof(ch));
    paintDC->TextOut(10, position, ch);
    position += tm.tmHeight;

    s2.GetStr(ch, sizeof(ch));
    paintDC->TextOut(10, position, ch);
    position += tm.tmHeight;

    s2.Insert("CAT ", 5);

```

```

s2.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s3 = "IN THE THE";
s3.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s2.Insert(s3, 9);
s2.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s2.Delete(16, 3);
s2.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s3 = s2;
s2.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

wsprintf(ch, "S3 is %d characters long.", s3.Length());
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

wsprintf(ch, "'CAT' is at position %d.", s3.Pos("CAT"));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s2 = "HAT";
wsprintf(ch, "'HAT' is at position %d.", s3.Pos(s2));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s3 = s2 + " TRICKS";
s3.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s3 = s3 + " " + s3;
s3.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s1 = s3.GetSubStr(5, 6);
s1.GetStr(ch, sizeof(ch));
paintDC->TextOut(10, position, ch);
position += tm.tmHeight;

s3 = s2;

```



```
    if ((s2 == "HAT") && (s2 == s3))
        paintDC->TextOut(10, position, "The strings are equal.");
}
```

---

The output from the test program should look like Figure 35.1.

Now not only do you have a handy programming tool, but also you have reinforced some of what you learned earlier in this chapter—specifically, what you learned about the proper use of function and operator overloading. This `String` class overloads functions that vary in parameter type, not parameter count. Also, it overloads operators in a way that is consistent with their intended use—except in one instance. Do you see a problem with the concatenation function? The concatenation function will allow an expression like

```
str3 = str1 + str2
```

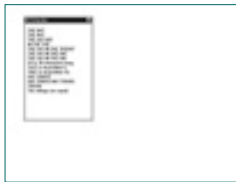
or

```
str3 = str1 + "TEST"
```

It won't, however, allow an expression like

```
str3 = "TEST" + str1
```

Why? Because, if you recall, it's the object on the left of the operator that calls the overloaded operator function. Because "TEST" is a character array and not a `String` object, the previous expression is invalid. It won't even compile. To perform the operation in question, "TEST" must first be converted to a `String` object.



**FIG. 35.1** The STRGAPP application gives the *String* class a chance to perform its tricks.

## Conclusion

Over the course of this book, you've learned a great deal about Microsoft Foundation Classes, everything from creating an AppWizard application to writing MFC programs from scratch—including designing your own classes. You've learned about OLE and database applications. You've even dipped into technical subjects like linked lists and recursion. At this point, you should be well armed to get started on your own MFC projects. Whether you opt for the AppWizard approach to MFC programming, or you like the direct approach of writing all your code by hand, MFC gives you the tools that you need to create sensational Windows programs. So what are you waiting for? Get to it!

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH](#)  
ITKNOWLEDGE[Brief](#)   [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

[BROWSE](#)  
BY TOPIC

[an error occurred while processing this directive]

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

## Index

### Symbols

---

**<< and >> operators, saving, 57**

**<OBJECT> tags**

listings, 605

parameters, 605

### A

---

**About commands, changing, help menus, 45**

**accelerator, editors, 341-348**

**accepting drags, 698-699**

**access methods, friends, purpose, 149**

**accessing**

arrays, indexes, 180

CCntIDlg, limiting, 149

data members

    dialog boxes, 148

    document classes, 304

databases, relational, 452

member functions, pointers, 150

objects, mutexes, 412

pointers, variables, 43

**accessors**

get\_CaptionProp, listings, 648

transferring, TextDataPath, 673

**accommodating namespaces, user-defined, 445**

**acquiring data, clipboards, 690**

**activating**

attributes, fonts, 84

controls

    connection points, 626

flicker-free, 706

windowless, 705-706

**Active Template Library, *see* ATL**

**ActiveX**

applications, new, 518

automation servers, 520

basic calculators

creating, 589

running, 590-591

constructors

features, 558

listings, 558

container applications, 521

compiling, 522

definition, 519-520

objects, 523-524

source codes, 521-522

control bars, managing, 559

controls

address, 588

ATL, 629

creating, 588-589

definition, 520

overview, 587

properties, 643-644

Web pages, 604

*see also* controls

data, drawing, 560-567

initializing, listings, 574

objects

deleting, 547-549

selecting, 539-540

OLE, comparing, 518

server applications

compiling, 552, 566

creating, 551-554

definition, 520

functionality, 554

registering, 552, 555

running, 566

- testing, 553
- terminology, 518
- view classes, 558

## **ActiveXCont1**

- applications, compiling, 539
- CntrlItem classes, changing, 541-542
- features, 526

## **Add Method to Interface dialog boxes, 637**

### **adding**

- buttons, notification functions, 600-601
- cells, next to the lists, 750
- codes
  - document classes, 48-50
  - ShowDib, 345-346
- combo boxes, toolbars, 229
- control implementations, ATL, 631
- controls
  - Developer Studio's dialog box editor, 140
  - rich edit, 273-275
  - Web pages, 604
- custom panes, status bars, 235
- data members, CApp1Doc, 39
- dialog boxes, 118
- elements, arrays, 369-370
- entries, property pages, 656
- enumerations, dispIDTextDataPath, 671-672
- events, ReadyStateChange, 670
- functionality, dialog boxes, 18
- IDs, indicators arrays, 236
- interfaces
  - CProxyDATLControlWin, 659
  - events, 657
- member variables
  - listings, 672
  - Object Wizard, 649
- methods, controls, 602-603, 637
- new panes IDs, indicator arrays, 236
- nodes, lists, 375
- objects, CPoint, 42

- persistence, 674-675
- points, point arrays, 52
- progress bars, 165-168
- properties
  - asynchronous, 667-668
  - controls, 594
  - persistence, 661
  - ReadyState, 668-669
  - stock, 649
- property sheets
  - message handlers, 653-654
  - objects, 652-653
- records, databases, 463-468
- support, menus, 93-94

**addition controls, basic calculators, 608**

**additional items, creating, 195**

**address controls, ActiveX, 588**

**AddString( ) member functions, purpose, 151**

**advancing, animate applications, 285**

**advantages**

- ATL, 666
- persistence, 661

**AfxOleInit( ), calling, listings, 527, 554**

**aliases, namespaces**

- features, 447
- procedures, 447-448

**aligning paragraphs, Rich Edit, 272**

**alignment**

- enumerations, listings, 639
- member variables, 645
- parameters
  - listings, 641
  - ShowCaption, 641
- properties, listings, 646
- settings
  - include files, 639
  - enumerations, 638

**allocating spaces dynamically, 806**

**ambient**

- fonts, 663

properties, 651

## **animate applications**

advancing, 285

compiling, 283

exploring, 284

features, 278

reversing, 285

source codes, 278

## **Animate2 applications**

compiling, 290

exploring, 290

## **animations**

control

AVI files, 292

creating, 284

definition, 277

dialog boxes, 286-290

manipulating, 284

transparent style, 291

sequences

finding, 277

loading, 285

Windows 95, 277

## **API (Application Programming Interfaces)**

definition, 8

tools, learning, 8

functions, windows, 359

## **app 1 classes, functions, 38**

## **appearances**

image list controls, 180

menus, commands, 113

modifying, panels, 237-238

## **application classes, Life, listings, 721-730**

## **application information, 480**

## **application objects**

deriving, CWinApp, 10

instantiating, 63

## **Application Programming Interfaces, *see* API**

## **applications**

ActiveX, new, 518

- animate, 278
- autoserver
  - creating, 570-572
  - GUID, 574
  - source codes, 570
  - stand-alone, 573
- building, threads, 396-398
- C++, exception objects, 420
- Car 1
  - functions, 787
  - output, 788
  - version one, 786-787
- Car 2, listings, 788-790
- Car 3, source codes, 794
- colors, setting, 353
- compiling
  - ActiveXCont1, 539
  - bitmaps, 317
  - FTPApp, 507
  - rich edit, 275
  - threads, 410
- creating
  - bitmaps, 312
  - manually, 62
  - ShowDib, 338-341
- dialog boxes
  - functions, 124
  - header files, 124
  - implementation files, 125
  - overview, 122-124
- FTP, programming, 498-501
- FTPApp, 508
- HTTPApp, 494-495
- linking, device drivers, 74
- recursion, examples, 762-777
- running
  - MFC, 36-37
  - threads, 410
- SDI, window classes, 14
- threads, CCountArray2, 413



Web

    compiling, 493

    programming, 488-492

windows, clicking, 75

*see also* MFC App2 applications; console applications; container applications

**applications windows, repainting, listings, 51**

**AppWizard**

    data members, declaring, 40

    databases, creating, 452-453

    files, generating, 37-38

    listings, CApp1Doc, 40

    MFCRichEdit, procedures, 273-276

    settings, RichEdit, 263-264

    toolbars, creating, 207

**archives, creating, 310**

**areas, updating, bitmaps, 323**

**arguments**

    BitBlt() types, 321

    Create(), 65

    DIBs, types, 351

    SetButtonInfo(), 229

    TextOut(), 86

*see also* default arguments

**arrays**

    accessing, indexes, 180

    classes

        features, 365

        member functions, 366

    declaring, 369

    demo applications

        features, 367-369

        OnDraw() functions, 370

    elements

        adding, 369-370

        removing, 371

    initializing, 369

    operators, defining, 804

    reading, 370-371

**assigning IDs, Developer Studio, 182**

**assignment operators, character arrays, 811**

**associating image lists, list view, 192**

**asynchronous properties, adding, 667-668**

**ATL (Active Template Library)**

advantages, 666

COM AppWizard

choices, 619

procedures, 619

comparing, MFC, 618-619

control implementations, adding, 631

control projects, creating, 630

controls

ActiveX, 629

building, 634

creating, 630

registering, 634

dialog boxes, Object Wizard, 616

disadvantages, 617

implementing, 614

methods, creating, 637

Object Wizards, 620

optimized drawing capabilities, 679

overview, 616-617

projects

creating, 619

types, 618

properties, asynchronous, 667-668

source codes, locating, 614

**ATL Object Wizard Properties dialog boxes, properties, defining, 632**

**ATLCONTROLWIN.CPP parameters, listings, 640**

**attributes**

controls, summaries, 632-633

fonts, activating, 84

parameters

flowing, 638

IDL, 638

purpose, 638

**automatically generating IDLs, 616**

**automation**

- client applications
  - creating, 579
  - running, 585
  - source codes, 580
- initializing, 576
- OLE, 569
- servers
  - ActiveX, 520
  - codes, 555
  - creating, 569-570
  - methods, 576-579
  - properties, 576-579

#### **autoserver applications**

- creating, 570-572
- GUID, 574
- source codes, 570
- stand-alone, 573

#### **AVI files, animations, control, 292**

## **B**

---

**BackColor properties, dispid, 650**

**Barrec Classes, header files, 800**

**base class versions, calling, OnInitDialog(), 150**

**base classes, PreCreateWindow(), calling, 91**

#### **basic calculators**

- ActiveX, creating, 589
- controls
  - addition, 608
  - displaying, 607
  - multiplication, 609
- running, ActiveX, 590-591

**biClrUsed members, checking, BITMAPINFOHEADER structures, 338**

#### **binary trees**

- creating, 778-779
- definition, 777

**biSizeImage, data members, BITMAPINFOHEADER structures, 337**

**BitBlt(), arguments, types, 321**

**BITMAPFILEHEADER Structures**

- components, 326

- listings, 326

## **BITMAPINFO Structures, DIBs, listings, 327**

### **BITMAPINFOHEADER Structures**

- biClrUsed members, checking, 338

- components, 328

- data members, biSizeImage, 337

- DIBs, listings, 327

- size, controlling, 327

- values, calculating, 328

### **bitmaps**

- applications

  - compiling, 317

  - creating, 312

- areas, updating, 323

- checking, listings, 336

- functions

  - CreateCompatibleDC(), 319

  - OnCreate(), 319

  - SelectObject(), 320

- Life

  - features, 744

  - starting images, 744

- overview, 311

- pointers, 739

- windows, updating, 319

- see also* DIBs; DDBs

## **black box routines, structured-programming, 785**

### **blocks, creating catch programs, 422**

### **BMP formats, 325**

### **browsing, FTP servers, 510-511**

### **brushes**

- choosing, Life, 746

- creating, FillRect(), 320

- deselecting, 89

- features, 87

## **build commands, Developer Studio, 233**

### **building**

- applications, threads, 396-398

- controls, ATL, 634

**bunny-in-barrels, programs, listings, 757**

**ButtonCaption, properties, 596**

**buttons**

adding, notification functions, 600-601

controls, testing, 601-602

DLL option, procedures, 630

responding

toolbars, 216

wizards, 258

wizards

listings, 258

setting, 257

## **C**

---

**C++**

applications, exception objects, 420

exceptions, 421

overview, 419

RTTI, purpose, 434

**cached variables, cleaning, 680**

**caching, drawing resources, optimized drawing, 680**

**CActiveXCont1**

CntrlItem classes, member functions, 527-528

document classes

constructors, 532

features, 531

message maps, 532

view classes, 528

**CActiveXServSrvrItem classes, 556**

**calculating**

squares, locating, 322

time spans, listings, 390

values, BITMAPINFOHEADER structures, 328

**calling**

AfxOleInit(), listings, 527, 554

base classes

OnInitDialog(), versions, 150

PreCreateWindow(), 91

constructors, CFile, 308

member functions, GetCurSel(), 154

- OnData, listings, 674
- OnNewDocument() functions, 55-56
- ShowCaption, methods, 640
- cancel buttons, dialog boxes, 152-155**
- CAnimateCtrl, member functions, 285**
- capabilities, optimized drawing, ATL, 679**
- CApp1Doc**
  - AppWizard, listings, 40
  - data members, adding, 39
  - functions, app 1, 38
  - virtual member functions, types, 41
- CApp1View**
  - functions, app 1, 38
  - pointers, obtaining, 41-42
- Car 1 applications**
  - functions, 787-788
  - output, 788
  - versions, one, 786-787
- Car 2 applications, listings, 788-790**
- Car3 applications, source codes, 794**
- CArchive**
  - features, 27
  - objects, creating, 310
- casting, listings, 441**
- catch blocks**
  - creating, 422
  - exceptions, 424
  - multiple, 425-426
  - ordering, 424
  - parameters, 421
- catching exceptions, listings, 423**
- Categories**
  - classes, MFC, 9
  - constants, types, 479
  - globals, types, 479
  - macros, types, 479
  - properties, ActiveX controls, 643-644
- CAutoServerApp, classes, exploring, 574-575**
- CBase**
  - defining, 437

member functions, Func2(), 440

pointers, creating, 442

## **CBitmap, member functions, 319**

### **CBitmapView**

constant declarations, 317

functions, listings, 315-316

## **CBrush Objects, creating, listings, 88**

## **CChildFrame, functions, app 1, 38**

## **CClientDC, functions, 23**

## **CCmdTarget, classes, 10**

### **CCntDlg**

accessing, limiting, 149

radio buttons, initializing, 150

source code files, 146

### **CCountArray**

header files, listings, 407

impementation files, 407

### **CCountArray2**

applications, threads, 413

header files, listings, 411

## **CDC classes, deriving, 23-24**

### **CDialog**

classes, types, 18-20

features, 118

member functions, 19

### **CDib**

classes

features, 329

implementing, 331-335

constructors

definition, 330

listings, 335

data members, listings, 330

destructors, 330

header files, listings, 329-330

member functions

listings, 330

public, 331

types, 337-338

objects, creating, 335

## **CDlg1**

classes

header files, 128

implementation files, 129

objects, constructing, 133

parameters, 133

## **CDocument**

data, storing, 14-15

DeleteContents(), overriding, 348

## **cells**

adding, next to die lists, 750

creating, 710-711

killing, 710-711

Life, displaying, 746

neighbor counts, incrementing, 750

processing, listings, 751

requirements, Life, 752

speed, 711

## **CFile**

classes

file demo 3 applications, 306

member functions, 307

types, 26-27

constructors, calling, 308

files, opening, 310

objects, configuring, 26-27

## **CFont classes, fonts, 85**

## **CFrameWnd**

member functions, 14

public data members, 13

windows, 13

## **CFtpConnection**

member functions, 509

objects, creating, 509

## **changing**

About commands, help menu, 45

CntrlItem classes, ActiveXCont1, 541-542

display strings, listings, 308

enumerations, strings, 678-679

formats, paragraphs, 272-273



- list view, listings, 196
- OnDraw(), ShowDib, 359-360
- sizes, stacks, 761
- values, m\_display, 89

## **character arrays**

- assignment operators, 811
- constructors
  - listings, 810
  - source codes, 810
- flexibility, 806
- functions, Pos(), 814
- operators, concatenation, 812
- strings
  - destructing, 810
  - representing, 806

## **CHARFORMAT structures, listings, 271**

### **check boxes**

- controls, 140
- status, storing, 153

### **check marks, displaying, 114**

### **checking**

- biClrUsed members, BITMAPINFOHEADER structures, 338
- bitmaps, listings, 336

### **child items, tree view controls, listings, 205**

### **child windows, MDI, 16-17**

### **choosing brushes, Life, 746**

### **CHttpFile**

- member functions, types, 497
- objects, creating, 495-496

### **CImageList Class member functions, 183-184**

### **CInternetSession**

- member functions, 495-496
- objects, creating, 495

### **class constructors**

- member initialization, listings, 688
- member variables, 701-702

### **class derivation, listings, 439-440**

### **class member functions, wizards, 22**

### **class templates**

- collections, 382

- defining, listings, 430
- multiple parameters, listings, 433
- objects, instantiating, 430
- parameters, data types, 434
- procedures, listings, 431-432
- purpose, 430

## **classes**

- app 1, functions, 38
- arrays, 365
- CAutoServerApp, exploring, 574-575
- CCmdTarget, 10
- CDialog, types, 18-20
- CDib, 329
- CFile
  - file demo 3 applications, 306
  - member functions, 307
  - types, 26-27
- CList, templates, 615
- CMessages
  - deriving, 302
  - features, 301-302
  - procedures, 303-304
- CObject, 9-10
- constructors, OOP, 798
- controls
  - implementations, 626
  - types, 24-26
- creating
  - exceptions, 421
  - general-to-specific, 801
  - property pages, 248-249
  - property sheets, 248-249
  - virtual functions, 805
- custom, developing, 10
- CWinThread, 10
- DAO, overview, 449
- database, 27
- deriving, CDC, 23-24
- dialog boxes
  - features, 118

- procedures, 135
- fonts, CFont, 85
- functionality, determining, 806
- implementing, CDib, 331-335
- lists, 371-372
- maps
  - features, 378
  - functions, 379
- member functions
  - arrays, 366
  - lists, 372
- MFC
  - categories, 9
  - features, 8-9
  - message-mapping, 11-12
  - types, 29
- objects, defining, 802
- ODBC, overview, 449
- OOP
  - data types, 791
  - general, 797
  - problems, 797
- operators, CPoint, 391
- point
  - header files, 798
  - implementation files, 798
- rectangle
  - header files, 799
  - implementation files, 799
- shape
  - header files, 798
  - test programs, 799
- single-instance
  - appropriateness, 802
  - problems, 802
- specialized, CView, 15
- string
  - description, 808
  - features, 805
  - functions, 807

header files, 807

storage, 806

testing, strings, 816

utilities, overview, 385

WinInet, types, 487-488

writing, dialog boxes, 130-131

*see also* windows classes

**ClassWizard, delimiters, types, 480-481**

**clean documents, 60**

**cleaning**

lists, 377-378

variables, cached, 680

**cleanup, OnDestroy, listings, 681**

**Clear( ) functions, File, 753**

**clicking**

mice, OnLButtonDown( ), 322

windows applications, 75

**client applications**

automation, running, 585

creating, 579

source codes, 580

**client areas**

data, displaying, 74

invalidating, WM\_PAINT messages, 81

**clipboards**

data

acquiring, 690

placing, 687

transfers, 688

editing keys, 681

interfaces

COM, 684

requirements, 683

formats, customizing, 700

methods, OnKeyDown, 682

support

features, 681

listings, 687

types, 683

**CList**

- classes, templates, 615
- instances, creating, 615
- clsid arguments, GUID, 555**
- CLSIDs, 605**
- CMainFrame**
  - functions, app 1, 38
  - header files, listings, 210-211
  - implementation files, listings, 64-65, 211-213
- CMainFrameClass, header files, listings, 64**
- CMDIChildWnd**
  - features, 16
  - member functions, 17
- CMDIChildWnd classes, functionality, OOP, 17**
- CMDIFrameWnd**
  - documents, running simultaneously, 16
  - member functions, 16
- CMessages**
  - classes
    - deriving, 302
    - features, 301-302
    - procedures, 303-304
  - header files, listings, 301-302
  - implementation files, listings, 302-303
- CMFCApp1, constructors, 63**
- CMyToolbar**
  - header files, listings, 227
  - implementation files, listings, 227-229
- CNode structures, listings, 374**
- CntrItem classes**
  - CActiveXCont1, member functions, 527-528
  - changing, ActiveXCont1, 541-542
- CObject**
  - custom classes, developing, 10
  - data, serializing, 10
  - member functions
    - overview, 9
    - types, 10
- CObject classes, 9-10**
- codes**
  - adding

- document classes, 48-50
- ShowDib, 345-346
- automation servers, listings, 555
- exception-handling, placing, 420
- Format( ), 388-389
- generating, Object Wizard, 624
- OnFileOpen( ), listings, 347
- OnPaletteChanged( ), listings, 362
- OnQueryNewPalette( ), listings, 363-364
- optimized drawing, 706
- ShowDib, version 3, 362
- coding left button clicks, listings, 52**
- collection classes**
  - defining, 481
  - helper functions, 481
- collections, class templates, 382**
- color tables, pointers, listings, 338**
- COLORREF values, setting, 88**
- colors**
  - displaying, monitors, 74
  - lines, RGB macro, 86
  - mapping, windows, 353
  - numbers, listings, 338
  - setting, applications, 353
  - ShowDib, setting, 352
- columns, creating, list view, 192-193**
- COM (Component Object Model)**
  - features, 517
  - interfaces, clipboards, 684
  - OLE, comparing, 518
  - overview, 613
  - properties, setting, 621
- COM AppWizard, ATL**
  - choices, 619
  - procedures, 619
- combo boxes**
  - adding, toolbars, 229
  - controls, 140
  - creating, listings, 230
  - initializing, listings, 152, 230

## **command IDs**

consecutive, 111

creating, 235

responding, 113

*see also* IDs

## **command IU functions, toolbars, listings, 221**

## **command update handlers, creating, 236**

## **Command-UI Response Functions, Life, listings, 740**

## **commands**

appearances, updating, 113

build, Developer Studio, 233

controls, compiling, 634

enabling, menu applications, 101

help menu, About, 45

menus, check marks, 114

responding, 112

updating, menus, 114

## **communicating**

event objects, threads, 403-406

global variables, 399-401

overview, 117

user-defined messages, 401-403

## **comparing**

### **MFC**

ATL, 618-619

Windows, 8

ODBC, DAO, 477-478

### **OLE**

ActiveX, 518

COM, 518

parameters, 813

strings, operators, 812-813

windows, objects, 43

## **compatibility, DCs, 319**

## **compiling**

animate applications, 283

Animate2 applications, 290

applications

ActiveX server applications, 566

ActiveXCont1, 539

animate, 283, 290

bitmaps, 317

FTPApp, 507

rich edit, 275

threads, 410

Web, 493

console applications, 419

container applications, ActiveX, 522

controls

commands, 634

Windows 95, 164

MFC App2 applications, 71

programs manually, 66-68

RichEdit applications, 267

server applications, ActiveX, 552

user interfaces, 593

## **Component Object Model, *see* COM**

### **components**

BITMAPFILEHEADER Structures, 326-328

dwEffects, 271

dwMask, 271

## **computers, programming, introduction, 7**

### **concatenation**

functions, problems, 820

operators

character arrays, 812

string objects, 811

parameters, 812

strings, definition, 811

## **conditional statements, recursion, 761**

### **configuring**

connection points, interfaces, 670-671

demo applications, property sheets, 249-252

objects, CFile, 26-27

sliders, 174

spinners, 178

### **connection points**

controls, activating, 626

interfaces, configuring, 670-671

## **consecutive, command IDs, 111**



## **console applications**

compiling, 419

definition, 419

running, 419

*see also* applications

## **constant declarations, CBitmapView, listings, 317**

### **constants**

categories, 479

data, 482

dispid

IDL, 647

listings, 648

files, resource, 97-98

predefined, 479

threads, priorities, 396

## **constraining drags, 699**

### **constructing**

CDlg1 Objects, listings, 133

strings, 810

### **constructors**

ActiveX

features, 558

listings, 558

CActiveXCont1, document classes, 532

calling, CFile, 308

CDib, 330

character arrays

listings, 810

source codes, 810

classes, OOP, 798

CMFCApp1, 63

CPaintDC, 82

CString, 386

dialog boxes, 132

member variables, initializing, 662

view classes, listings, 502

## **container applications**

ActiveX, 521

compiling, ActiveX, 522

editing

inplace, 524

items, 526

objects

embedding, 523-524

linking, 524-526

refreshing, FireOnChanged, 646

source codes, ActiveX, 521-522

## **contents**

directories, getting, 511

linked lists, displaying, 717

**control bars, managing, ActiveX, 559**

**Control Dialog dialog boxes, controls, finding, 142**

**control IDs, creating, 121**

**control implementations, adding, ATL, 631**

**control projects, creating, ATL, 630**

## **controlling**

dialog boxes, adding, 118

size, BITMAPINFOHEADER structures, 327

styles, lines, 87

## **controls**

activating

connection points, 626

flicker-free, 706

windowless, 705-706

ActiveX

address, 588

ATL, 629

creating, 588-589

definition, 520

overview, 587

adding

Developer Studio's dialog box editor, 140

rich edit, 273-275

attributes, summaries, 632-633

basic calculators

addition, 608

creating, 589

multiplication, 609

running, 590-591

building ATL, 634

- buttons, testing, 601-602
- Check boxes, 140
- classes
  - implementations, 626
  - types, 24-26
- combo boxes, 140
- compiling
  - commands, 634
  - Windows 95, 164
- creating
  - ATL, 630
  - Rich Edit, 268
  - Windows 95, 159-164
- dialog boxes, types, 139
- displaying basic calculators, 607
- drawing, 661-662
- dual-interface
  - features, 705
  - methods, 622
- finding, Control Dialog dialog boxes, 142
- group boxes, 140
- image lists
  - appearances, 180
  - features, 179
- initializing
  - procedures, 170
  - Rich Edit, 268-270
- Internet Explorer, requirements, 632
- linking
  - pages, 656
  - Windows 95, 164
- list boxes, 140
- list view
  - Explorer, 185
  - features, 184
  - notifications, 196-197
- loading, TSTCON32, 636
- manipulating
  - purpose, 149
  - Rich Edit, 268-270

member functions, features, 149

methods

- adding, 602-603, 637

- testing, 603-604

OLE, definition, 517

progress bars

- adding, 165-168

- creating, 169

- initializing, 169-170

- manipulating, 170

- procedures, 168

properties

- adding, 594

- testing, 597-598

- updating, 655

- values, 597

Pushbuttons, 140

radio buttons, 140

redrawing, listings, 643

registering

- ATL, 634

- manually, 635

scroll bars, 140

sliders

- configuring, 174

- features, 170-178

- initializing, 173

- manipulating, 175

- responding, 175

- styles, 173

spinners

- configuring, 178

- creating, 177-178

- features, 175

- initializing, 177

- styles, 177

tabs

- Miscellaneous, 633

- Stock Properties, 633

testing, TSTCON32, 636

- tree view
  - features, 198
  - styles, 202
- updating, listings, 655-656
- user interfaces
  - compiling, 593
  - creating, 591-594
- Web pages, ActiveX, 604
- windows
  - history, 159
  - types, 139-140
- conventions, interfaces, naming, 625**
- converting, Variant types, listings, 642**
- Conway, John, simulations, rules, 709**
- CopyDataToClipboard helper functions, listings, 688-689**
- copying data, dialog boxes, 155**
- CopyStgMedium helper functions, listings, 690**
- correct data types, pointers, 337**
- counting resource access, semaphores, 413**
- CPaintDC**
  - constructors, 82
  - DCs, managing, 82
  - objects, deleting, 82
- CPen constructors, styles, defining, 87**
- CPoint**
  - classes, operators, 391
  - objects
    - adding, 42
    - creating, 390
    - uninitialized, 390
- CProgressCtrl member functions, 169**
- CPropertyPage member functions, types, 22**
- CPropertySheet**
  - data members, 21
  - member functions, 21-22
- CProxyDATLControlWin interfaces, adding, 659**
- Create()**
  - arguments, 65
  - prototypes, listings, 65
- CreateDibPalette( ) functions, listings, 356-357**

**CreateFontIndirect()**, fonts, creating, 85

**CreateListView()** functions, listings, 180-181, 186-187

**CreateProgressBar()** functions, listings, 165-166

**CreateSlider()** functions, listings, 171

**CreateSpinner()** functions, listings, 176

**CREATESTRUCT**

listings, 91

members, types, 91

**CreateTreeView()** functions, listings, 180-181, 199-200

**creating**

animations, control, 284

applications

autoserver, 570-572

bitmaps, 312

manually, 62

ShowDib, 338-341

basic calculators, ActiveX, 589

blocks, catch programs, 422

brushes, FillRect(), 320

CBrush Objects, listings, 88

cells, 710-711

classes

exceptions, 421

general-to-specific, 801

property pages, 248-249

property sheets, 248-249

virtual functions, 805

client applications, automations, 579

columns, list view, 192-193

combo boxes, listings, 230

command IDs, 235

command update handlers, 236

control IDs, 121

control projects, ATL, 630

controls

ActiveX, 588-589

ATL, 630

Rich Edit, 268

user interfaces, 591-594

Windows 95, 159-164

- critical sections, threads, 406
- data members, dialog boxes, 18
- databases
  - AppWizard, 452-453
  - display, 460-463
  - ODBC, 453
- default strings, 235
- dialog boxes
  - procedures, 121
  - resource editors, 18-20
- drawing objects, optimized drawing, 679
- employee applications, 455-459
- fonts
  - CreateFontIndirect(), 85
  - Paint1, 85
- image lists, 182-183
- indicator arrays, status bars, 234
- indicator IDs, listings, 234
- instances, CList, 615
- items
  - additional, 195
  - list view, 194-196
  - tree view, 203-205
- linked lists, 712
- list view, 189
- logical palettes, 359
- maps, 380
- memory DCs, OnDraw(), 321
- message response functions, toolbars, 215-217
- methods, ATL, 637
- nodes, listings, 374
- objects
  - CArchive, 310
  - CDib, 335
  - CFtpConnection, 509
  - CHttpFile, 495-496
  - CInternetSession, 495
  - CPoint, 390
  - CRect, 391
  - CSize, 391

- persistent, 300
- pages, wizards, 257
- PaintDC, listings, 83
- persistent classes, 299
- pointers
  - CBase, 442
  - polymorphism, 795
- programs
  - approaches, 31
  - examples, 32-36
  - manually, 61
- progress bars, 169
- projects, ATL, 619
- properties
  - Get() property methods, 595
  - notification methods, 595
  - Set() property methods, 595
  - user-defined, 644-645
- property sheets
  - demo applications, 243-245
  - procedures, 20
- radio buttons, toolbars, 222-224
- rectangles
  - applications, 43-44
  - new, 801
- resources
  - dialog boxes, 119-120
  - menus, 94-96
  - property pages, 245-247
  - toolbars, 214-215
- RichEdit applications, 262-267
- root items, listings, 204-205
- semaphores, 413
- servers
  - applications, ActiveX, 551-554
  - automation, 569-570
- ShowDib, version 3, 361
- sliders, 173
- spinners, 177-178
- subitems



- columns, 193
- listings, 195
- templates
  - dialog boxes, 130
  - functions, 427
  - InitInstance(), 554
- toolbar applications, final versions, 238
- toolbars, 217-218
  - AppWizard, 207
  - custom, 224-226
- tree view, 201-202
- trees, binary, 778-779
- view buttons, listings, 196
- windows, Life, 741

## **CRect**

- member functions, types, 392
- objects
  - creating, 391
  - manipulating, 392
  - uninitialized, 392
- operators, 393

## **critical sections**

- creating, threads, 406
- ownership, releasing, 407

## **CSize objects**

- creating, 391
- uninitialized, 391

## **CSliderCtrl member functions, 173-174**

## **CSomeResource**

- header files, semaphores, 414
- implementation files, listings, 414-415

## **CSpinButtonCtrl member functions, 178**

## **CStatusBar methods, 231**

## **CString**

- constructors, 386
- functions, formatting, 482
- member functions, 385-386
- objects, 481
- operators, 386

## **CTime**

- member functions, 387

- objects, 388

### **CTimeSpan member functions**

- objects, 390

- types, 387

### **CToolBarApp**

- header files, 209

- implementation files, 210

### **custom classes, developing, CObject, 10**

### **custom data format member variables, listings, 701**

### **custom formats, data, sending, 704**

### **custom panes**

- adding, status bars, 235

- update handlers, listings, 237

### **custom window classes, details, hiding, 8**

### **customizing**

- formats

  - clipboards, 700

  - Drag-and-Drop, 700

  - transferring, 701-702

- storage, document classes, 54

### **CView**

- classes, specialized, 15

- functionality, view windows, 14

- member functions, 15

### **CWinApp**

- application objects, deriving, 10

- member functions, 12

- public data members, 11-12

- services, 10

### **CWindowDC, functions, 23**

### **CWinThread, classes, 10**

### **CWnd base, window classes, deriving, 13**

## **D**

---

### **DAOs (Data Access Objects)**

- classes, overview, 449

- features, 27

- ODBC

  - comparing, 477-478

differences, 478

similarities, 478

## **data**

acquiring, clipboard, 690

CObject, serializing, 10

copying, dialog boxes, 155

displaying

client areas, 74

DCs, 23

OnDraw( ), 57

documents, initializing, 298

drawing, ActiveX, 560-567

encapsulating

hiding, 786

properties, 643

loading, classes, 26

losing, warnings, 60

moving, dialog boxes, 137

objects

initializing, 304

serializing, 305

placing, clipboards, 687

retrieving, listings, 137

returning, callers, 705

saving, classes, 26

sending, custom formats, 704

storing

CDocument, 14-15

rectangles applications, 54

transferring

DoDataExchange( ), 134

get\_TextDataPath, 672-673

listings, 136

types, constants, 482

**Data Access Objects, *see* DAOs**

## **data members**

accessing

dialog boxes, 148

document classes, 304

adding, CApp1Doc, 39

biSizeImage, BITMAPINFOHEADER structures, 337

CDib, listings, 330

CPropertySheet, 21

creating, dialog boxes, 18

declarations, listings, 189, 507

declaring

    AppWizard, 40

    dialog boxes, 150

dialog boxes

    listings, 131

    public, 131

displaying, view objects, 41

encapsulation objects, 790

initializing, listings, 136

modifying, view objects, 41

protection, declaring, 132

**data strings, editing, listings, 304-305**

**data transfers, initiating, clipboards, 688**

**data types**

    classes

        OOP, 791

        templates, parameters, 434

    correct, pointers, 337

**Database Management Systems, *see* DBMS**

**databases**

    creating

        AppWizard, 452-453

        ODBC, 453

    displaying, 460-463

    flat models, 450

    overview, 449

    records

        adding, 463-468

        deleting, 463-468

        filtering, 470-473

        sorting, 470-473

    registering procedures, 453-455

    relational

        accessing, 452

        definition, 450-452

SQL, 452

**DBMS (Database Management Systems), 450**

**DCs (Device Contexts), 74**

compatibility, 319

data, displaying, 23

definition, 23

graphical objects, new, 85

managing, CPaintDC, 82

**DDBs (Device-Dependent Bitmaps)**

definition, 311

reviewing, 325

*see also* bitmaps

**DDV Functions, types, 135**

**DDX Functions, types, 134**

**declarations**

data members, listings, 189, 507

message maps, listings, 189

**declaring**

arrays, 369

data members

AppWizard, 40

dialog boxes, 150

protection, 132

document data, 54

lists, 374-375

message handlers, menus, 109

message maps, manually, 68

update-command-UI functions, 109-110

**default, messages, file demo applications, 296**

**default arguments, overloading, differences, 802**

**default menu bars, modifying, menu editors, 45**

**default states, radio buttons, 151**

**default strings, creating, 235**

**defining**

CBase, 437

class templates, 430

collection classes, 481

events, methods, 658

GUID, 575

IDs, 96

- items, menus, 97
- message maps, manually, 68-69
- namespaces
  - listings, 445
  - unnamed, 447
- objects
  - classes, 802
  - OOP, 790
- operators, arrays, 804
- scopes, namespaces, 444-445
- styles, CPen constructors, 87
- templates, 427
- update-command-UI functions, 111

**definitions, nesting, namespaces, 446**

**DeleteContents()**

- features, 55
- overriding CDocument, 348

**deleting**

- GDI objects, 86
- logical palettes, purpose, 360
- nodes, lists, 375-376
- objects
  - ActiveX, 547-549
  - CPaintDC, 82
- records, databases, 463-468
- strings, 815

**delimiters, ClassWizard, types, 480-481**

**demo applications**

- arrays, 367-369
- configuring property sheets, 249-252
- lists, 372-374
- maps
  - features, 379-380
  - OnLButtonDown( ), 381
- property sheets
  - creating, 243-245
  - options, 243
  - source codes, 243
- running
  - property sheets, 252-253

wizards, 255-256

**demonstrations, linked lists, 713**

**deriving**

application objects, CWinApp, 10

classes

CDC, 23-24

CMessages, 302

window classes, CWnd base, 13

**deselecting**

brushes, 89

GDI objects, 86

**destructing strings, 810**

**destructors, CDib, 330**

**details, hiding, custom window classes, 8**

**determining**

functionality, classes, 806

general classes, OOP, 797

lfHeight fields, fonts, 84

sizes, image data, 328

**Developer Studio**

commands, build, 233

IDs, assigning, 182

**Developer Studio's dialog box editor, controls, adding, 140**

**developing, custom classes, CObject, 10**

**device contexts**

overview, 73, 351

*see also* DCs

**device drivers, applications, linking, 74**

**Device-Dependent Bitmaps, *see* DDBs**

**Device-Independent Bitmaps, *see* DIBs**

**diagnostic services, types, 483-484**

**dialog box classes**

header files, 146

implementation files, 147-148

**dialog boxes**

Add Method to Interface, 637

adding

controlling, 118

procedures, 118

animations, control, 286-290

- applications
  - functions, 124
  - header files, 124
  - implementation files, 125
  - overview, 122-124
- ATL Object Wizard Properties, 632
- cancel buttons, 152-155
- classes
  - features, 118
  - procedures, 135
  - writing, 130-131
- constructors, 132
- Control Dialog, 142
- controls
  - types, 139
  - see also* controls
- creating
  - procedures, 121
  - resource editors, 18-20
- data
  - copying, 155
  - moving, 137
- data members
  - accessing, 148
  - creating, 18
  - declaring, 150
  - listings, 131
  - public, 131
- displaying
  - file demo 3 applications, 308
  - OnDialogTest( ), 136
  - procedures, 129-130
  - window classes, 154
- functionality, adding, 18
- MFC, 18
- Object Wizard, ATL, 616
- OK buttons, 152-155
- overview, 117
- Properties, check boxes, 622
- removing, procedures, 154



- resource scripts, 121-122
- resources, creating, 119-120
- storage, listings, 135
- templates, creating, 130
- see also* property sheets

## **DIBs (Device-Independent Bitmaps)**

- arguments, types, 351
- BITMAPINFO Structures, listings, 327
- BITMAPINFOHEADER Structures, listings, 327
- definition, 311
- displaying procedures, 338-341
- memory, reading, 336
- reviewing, 325
- RGBQUAD Structures, 328
- structures, types, 326

## **differences, DAO, ODBC, 478**

## **directories**

- contents, getting, 511
- source codes, Paint 1, 76

## **dirty documents, 60**

## **disadvantages, ATL, 617**

## **dispatch maps, document classes, listings, 576**

## **dispid**

- constants
  - IDL, 647
  - listings, 648
- enumerations, listings, 645
- properties, BackColor, 650

## **dispidTextDataPath enumerations, adding, 671-672**

## **display strings, changing, listings, 308**

## **displaying**

- cells, Life, 746
- check marks, 114
- colors, monitors, 74
- controls, basic calculators, 607
- data
  - client areas, 74
  - DCs, 23
  - OnDraw( ), 57
- data members, view objects, 41

- dialog boxes
  - file demo 3 applications, 308
  - OnDialogTest(), 136
  - procedures, 129-130
  - window classes, 154
- DIBs, 338-341
- linked lists, contents, 717
- property sheets, 259
- servers, OLE, 575
- toolbars, 217-218
- windows
  - OnPaint(), 81
  - ResourceView, 44
- wizards, 259

**displays, creating, databases, 460-463**

**DLL (Dynamic Link Library), option buttons, 630**

**DoBlanks() functions, recursion, 781**

**document classes**

- CActiveXCont1, 531
- codes, adding, 48-50
- constructors, CActiveXCont1, 532
- customizing storage, 54
- data members, accessing, 304
- dispatch maps, listings, 576
- message maps, CActiveXCont1, 532
- modifying, 48-50

**document data**

- declaring, 54
- initializing, 54-55
- modifying, view objects, 58-59
- serializing, 56-57
- variables, 54

**document objects**

- features, 39
- responsibilities, 54

**document/view architecture**

- definition, 14
- overview, 39
- see also* view windows

**documents**

- clean, 60
- data, initializing, 298
- dirty, 60
- HTML
  - processing, 497
  - reading, 496-497
- running simultaneously, CMDIFrameWnd, 16

### **DoDataExchange()**

- data, transferring, 134
- functions, 132
- overriding, listings, 133

### **DoFilter() functions**

- examining, 475-477
- listings, 474

### **DoModal() functions, 137**

#### **downcasting**

- definition, 435
- dynamic\_cast
  - operators, 435
  - pointers, 442
- RTTI, 443-444

### **downloading files, FTP servers, 511-512**

#### **Drag-and-Drop**

- formats, customizing, 700
- sources, enabling, 694
- supporting, overview, 693
- targets, enabling, 696

### **dragging toolbars, 208**

#### **drags**

- accepting, 698-699
- constraining, 699
- initiating, requirements, 694

### **DrawGrid() functions, 320**

#### **drawing**

- controls, 661-662
- data, ActiveX, 560-567
- lines, pens, 87
- rectangles
  - listings, 216
- polymorphism, 800

- squares, 323
- text, fonts, 85

### **drawing objects**

- creating, optimized drawing, 679
- see also* objects

### **drawing resources, caching, optimized drawing, 680**

### **drawing-object classes, MFC, 24**

### **DrawSquare()**

- memory DCs, 322
- squares, placing, 322

### **dual-interface controls**

- features, 705
- methods, 622

### **dwEffects, components, 271**

### **dwMask, components, 271**

### **dynamic property enumerations, 675-676**

### **Dynamic Link Library, *see* DLL**

### **dynamic\_cast**

- operators, 435
- pointers, 442

### **dynamically allocated memories, strings**

- destructing, 810
- representing, 806

## **E**

---

### **EALIGNMENT enumerations, 639**

### **Edit boxes, 140**

### **editing**

- data strings, 304-305
- items, container applications, 526

### **editing keys, clipboards, 681**

### **editors**

- accelerator, procedures, 341-348
- see also* resource editors

### **elements, arrays**

- adding, 369-370
- removing, 371

### **embedding objects**

- container applications, 523-524
- definition, 516

**employee applications, creating, procedures, 455-459**

**enable menus, responding, 112**

**enabling**

commands, menu applications, 101

Drag-and-Drop

sources, 694

targets, 696

RTTI, 437-438

**encapsulating**

data, properties, 643

data members, objects, 790

definition, 786

obstacles, finding, 791

**entering parameters, exceptions, 424**

**entries, adding, property pages, 656**

**enumerations**

adding, dispidTextDataPath, 671-672

alignment settings

include files, 639

specifying, 638

changing, strings, 678-679

dispid, 645

EALIGNMENT, 639

properties

dynamic, 675-676

features, 675

string values, getting, 676

**enumerators, functions, 685**

**event objects, communicating, threads, 403-406**

**events**

adding, ReadyStateChange, 670

functions, FireChange, 660

interfaces

adding, 657

tweaking, 660

methods, defining, 658

proxy codes, generating, 658-659

responding, OLE, 657

**examining**

functions

- DoFilter( ), 475-477
- OnMove( ), 469
- OnRecordAdd( ), 468
- OnRecordDelete( ), 470
- OnSortDept( ), 475

Life functions

- OnPaletteChanged( ), 363-364

- OnQueryNewPalette( ), 364

### **examples**

- applications, recursion, 762-777

- program, creating, 32-36

- recursion functions, 756-757

- trees, traversing, 780

### **exception objects, applications, C++, 420**

### **exception-handling codes, placing, 420-421**

### **exceptions**

- C++, 421

- catching, 423

- classes, creating, 421

- handling, options, 426

- multiple, catch blocks, 424

- objects, types, 422

- parameters, entering, 424

- processing, 484

- throwing, functions, 424

### **expanding, templates, 614-615**

### **Explorer controls, list view, 185**

### **exploring**

- animate applications, 284

- Animate2 applications, 290

- applications

- FTPApp, 508

- HTTPApp, 494-495

- functions

- GetHitItem( ), 545-546

- SetSelectedItem( ), 546-547

- InitTracker( ) functions, 545

- OnDraw( ) functions, 544-545

- OnEditClear( ) functions, 550

- OnLButtonDbClick( ) member functions, 543

OnLButtonDown( ) functions, 542-543

OnSetCursor( ) functions, 543-544

rectangles applications, 53

### **exponentiation, recursion**

listings, 758

Power( ), 758

### **extracting, strings, 815-816**

## **F**

---

### **failed, polymorphism, 442**

### **fields, LOGFONT, 83**

#### **File functions**

Clear( ), 753

OnGenerations( ), 754

OnRandomCells( ), 753

OnSpeed( ), 754

SubNbrs( ), 753

### **file demo 2 applications, 300-301**

#### **file demo 3 applications**

CFile classes, 306

dialog boxes, displaying, 308

features, 305

starting, 307

#### **file demo applications**

features, 296

messages, default, 296

source codes, 297

### **file mode flags, types, 309**

#### **files**

AppWizard, generated, 37-38

AVI animations, 292

downloading, FTP servers, 511-512

opening, CFile, 310

reading

    directly, 305

    listings, 309-310

resource

    constants, 97-98

    features, 97

writing, directly, 305

- FillRect( ) brushes, creating, 320**
- filtering records, databases, 470-473**
- final versions, toolbar applications, creating, 238**
- finding**
  - animation sequences, 277
  - controls, Control Dialog dialog boxes, 142
  - obstacles, encapsulation, 791
  - property sheets, Windows 95, 241
- FireChange functions, events, 660**
- FireOnChanged containers, refreshing, 646**
- flags, tree view, types, 203**
- flat models, databases, definition, 450**
- flexibility, character arrays, 806**
- fonts**
  - ambient, properties, 663
  - attributes, activating, 84
  - classes, CFont, 85
  - creating
    - CreateFontIndirect( ), 85
    - Paint1, 85
  - GDI, overview, 83
  - lfHeight fields, determining, 84
  - text, drawing, 85
- Format()**
  - codes, types, 388-389
  - strings, 388
- formats**
  - BMP, 325
  - customizing
    - clipboards, 700
    - Drag-and-Drop, 700
    - transferring, 701-702
  - paragraphs, changing, 272-273
- formatting functions, CString, 482**
- frame windows, 13**
- frames, stacks, definition, 760**
- FTP applications, programming, 498-501**
- FTP servers**
  - browsing, 510-511
  - files, downloading, 511-512



- parameters, default values, 512
- root directories, reading, 509-510

## **FTPApp applications**

- compiling, 507
- exploring, 508

## **Func2() member functions, CBase, 440**

## **function templates, problems, 428**

## **functionality**

- ActiveX, server applications, 554
- adding, dialog boxes, 18
- CMDIChildWnd classes, OOP, 17
- determining, classes, 806
- OLE 2, 516
- view windows, CView, 14

## **functions**

- alignments, setting, 646
- API windows, 359
- Car 1 applications, 787-788
- CBitmapView, 315-316
- character arrays, Pos( ), 814
- classes, string, 807
- Clear( ), File, 753
- concatenation, problems, 820
- conditional statements, recursion, 761
- CreateDibPalette( )
  - features, 357
  - listings, 356
- CreateListView( ), 180-181, 186-187
- CreateProgressBar( ), 165-166
- CreateSlider( ), 171
- CreateSpinner( ), 176
- CreateTreeView( ), 180-181, 199-200
- declarations, 538
- DeleteContents( ), 348
- DoDataExchange( ), 132
- DoFilter( )
  - examining, 475-477
  - listings, 474
- DoModal( ), 137
- DrawGrid( ) brushes, 320

- enumerators, 685
- exceptions, throwing, 424
- FireChange, events, 660
- formatting, CString, 482
- GetHitItem( ), exploring, 545-546
- GetTextExtent, 663
- GlobalAllocPtr( ), 336
- helper, collection classes, 481
- InitInstance( ), 584
- InitTracker( ), exploring, 545
- Invalidate( ), 89
- message maps, writing, 70
- MoveTo( ), positioning lines, 87
- OnCentered( ), 265
- OnChangeItemPosition( ), 533
- OnConnectMakeconnection( ), 492-493, 501-502
- OnCreate( )
  - bitmaps, 319
  - listings, 314
- OnDraw( )
  - exploring, 544-545
  - listings, 251, 317, 348, 503
- OnEditClear( ), exploring, 550
- OnFileSave( ), 308
- OnGenerations( ), File, 754
- OnHScroll( ), 171-172
- OnInitDialog, 654-655
- OnLButtonDbIClk( ), 503-504, 535-536
- OnLButtonDown( )
  - exploring, 542-543
  - listings, 167, 315, 369
- OnLeft( ), 264
- OnMove( )
  - examining, 469
  - listings, 467
- OnNotify( ), 188
- OnPaletteChanged( ), examining, 363-364
- OnPropSheet( ), 252
- OnQueryNewPalette( ), examining, 364
- OnRandomCells( ), File, 753

- OnRButtonDown( ), 505-506
- OnRecordAdd( )
  - examining, 468
  - listings, 467
- OnRecordDelete( )
  - examining, 470
  - listings, 468
- OnSetCursor( )
  - exploring, 543-544
  - listings, 536-537
- OnSetSetstring( ), 565
- OnSortDept( )
  - examining, 475
  - listings, 473
- OnSortID( ), 473
- OnSortName( ), 473
- OnSortRate( ), 473
- OnSpeed( ), File, 754
- OnUnderlined( ), 264, 275
- overloading, definition, 802
- parameters, recursion, 760
- PreCreateWindow( ), overriding, 90
- PreviousDirectory( ), 506
- Rectangle( ), 88
- recursion
  - DoBlanks( ), 781
  - examples, 756-757
  - listings, 756
- SelectObject( )
  - bitmaps, 320
  - brushes, 88
- Serialize( ), document classes, 299
- SetButtonInfo( ), arguments, 229
- SetSelectedItem( ), exploring, 546-547
- ShowBrushes( ), 88
- StretchDIBits( ), 351
- strings, Pos( ), 813
- SubNbrs( ), File, 753
- templates
  - creating, 427

- defining, 427
- listings, 427-428
- ThreadProc( ), 398
- trees, traversing, 780-781
- virtual
  - definition, 805
  - implementing, 797
- Win 95 controls, 187-188
- see also* member functions

## G

---

### **GDI (Graphics Device Interfaces)**

- fonts, overview, 83
- functions, 23

### **GDI objects**

- deleting, 86
- deselecting, 86
- overview, 73

### **general classes, OOP, 797**

### **generated files, AppWizard, 37-38**

### **generating**

- codes, Object Wizard, 624
- IDL, automatically, 616
- messages, WM\_PAINT, 90
- proxy codes, 658-659
- values, setting, 754

### **Get( ) property methods, properties, creating, 595**

### **get\_CaptionProp, accessors, 648**

### **get\_TextDataPath, data, transferring, 672-673**

### **GetCurSel( ) member functions, calling, 154**

### **GetDataFromClipboard, implementations, 690-691**

### **GetDataFromTransfer, implementations, 691-692**

### **GetDataFromTransfer Update, 702-703**

### **GetDlgItem( )**

- member functions, procedures, 149
- return values, 150

### **GetHitItem( ) functions, exploring, 545-546**

### **GetLBText( ) member functions, 154**

### **GetPredefinedStrings, 677**

### **GetTextExtent functions, 663**

**getting nodes, linked lists, 749**

**global variables**

communicating, threads, 399-401

handling, 790

**GlobalAllocPtr( ) functions, 336**

**Globally Unique Identifiers, *see* GUIDs**

**globals, categories, types, 479**

**graphical objects, DCs, new, 85**

**Graphics Device Interfaces, *see* GDIs**

**group boxes, controls, 140**

**GUIDs (Globally Unique Identifiers)**

autoserver applications, 574

clsid arguments, 555

defining, 575

## **H**

---

**handling**

exceptions, options, 426

palettes, ShowDib, 355

strings, 806

variables

global, 790

OOP, 790

**header files**

Barrec Classes, 800

CCountArray, 407

CCountArray2, 411

CDib, 329-330

classes, strings, 807

CMainFrame, 210-211

CMessages, 301-302

CMyToolbar, 227

CSomeResource, semaphores, 414

CToolbarApp, 209

dialog box classes, 146

listings

CDlg1 Class, 128

CMainFrameClass, 64

including, 132

main window classes, 142-143

- maps, types, 625
- menu applications, 103-104
- MFC App2 applications, 70-71
- OOP, 791
- point classes, 798
- rectangle classes, 799
- shape classes, 798

## **help menu commands, About, changing, 45**

### **helper functions**

- collection classes, 481
- CopyDataToClipboard, 688-689
- CopyStgMedium, 690
- PrepareDataForTransfer, 689

### **hiding**

- data, encapsulating, 786
- details, custom window classes, 8
- toolbars, 221-222

## **horizontal lines, listings, 321**

### **HTML documents**

- processing, 497
- reading, 496-497

## **HTTPApp applications, exploring, 494-495**

## **I-J-K**

---

### **IDataObject interfaces, 684-685**

#### **IDL (Interface Definition Language)**

- constants, dispid, 647
- generating, automatically, 616
- parameters, attributes, 638

### **IDropSource interfaces, 694**

### **IDropTarget interfaces, 696-697, 700**

### **IDs**

- assigning, Developer Studio, 182
- defining, menus, 96
- indicators arrays, adding, 236
- predefined, menus, 96-97
- standard, widows, 486
- see also* command IDs

### **IEnumFORMATETC implementations**

- features, 685

listings, 686

**ignoring polymorphism, 442-444**

**image data, sizes, determining, 328**

**image lists**

associating, list view, 192

controls

appearances, 180

features, 179

creating, 182-183

initializing, 183-184

list view, 190

**IMPLEMENT\_SERIAL() macros, 303**

**implementation files**

CCountArray, 407

CMainFrame, 211-213

CMessages, 302-303

CMyToolbar, 227-229

CSomeResource, 414-415

CToolbarApp, 210

dialog box classes, 147-148

Life, 710

listings

CDlg1 Class, 129

CMainFrame, 64-65

MFC App2 applications, 71

main window classes, 143-146

menu applications, 104-108

OOP, 791

point classes, 798

rectangle classes, 799

**implementations**

controls, classes, 626

GetDataFromClipboard, 690-691

GetDataFromTransfer, 691-692

IEnumFORMATETC

features, 685

listings, 686

OnCreate, 698

OnDestroy, 698-699

OnKeyDown, 693

OnLButtonDown, 695

## **implementing**

ATL, 614

classes, CDib, 331-335

functions, virtual, 797

nodes, trees, 779

properties, ReadyState, 669

## **in-place editing, container applications, 524**

## **include files, enumerations, alignment settings, 639**

## **incrementing, neighbor counts, linked lists, 750**

## **indexes, arrays, accessing, 180**

## **indicator arrays**

creating, status bars, 234

new panes IDs, adding, 236

## **indicator IDs, creating, 234**

## **information, obtaining, typeid operators, 435**

## **inheritance, OOP, definition, 792**

## **initializing**

ActiveX, 574

arrays, 369

automation, OLE, 576

combo boxes, 152, 230

controls, 170

data

documents, 298

objects, 304

document data, 54-55

list boxes, 151

list view, 191

lists

image, 183-184

linked, 748

LOGFONT, 84

logical palettes, 359

LV\_COLUMN Structures, 193

LV\_ITEM Structures, 195

maps, 380

member variables

class constructors, 701-702

constructors, 662



- listings, 640
- m\_IReadyState, 669
- progress bars, 169-170
- radio buttons, CCntIDlg, 150
- Rich Edit, 268-270
- servers, OLE, 574-575
- sliders, 173
- spinners, 177
- tree view, 202
- variables
  - Life, 741
  - Variant, 642

### **initiating**

- data transfers, clipboards, 688
- drags, requirements, 694

### **InitInstance( )**

- functions, 584
- source codes, 554
- templates, creating, 554

### **InitTracker( ) functions, exploring, 545**

### **Insert New Object Commands, responding, 528-529**

### **inserting strings, 814-815**

### **instances, creating, CList, 615**

### **instantiating**

- application objects, 63
- objects, class templates, 430

### **Interface Definition Language, *see* IDL**

### **interfaces**

- adding
  - CProxyDATLControlWin, 659
  - events, 657
- clipboards
  - COM, 684
  - requirements, 683
- connection points, configuring, 670-671
- events, tweaking, 660
- IDataObject, 684-685
- IDropSource, 694
- IDropTarget, 696-697, 700
- naming, conventions, 625

**Internet Explorer controls, requirements, 632**

**initializing data members, 136**

**Invalidate( )**

functions, 89

repainting, 60

**invalidating client areas, WM\_PAINT messages, 81**

**items**

creating

additional, 195

list view, 194-196

tree view, 203-205

defining, menus, 97

editing, container applications, 526

list view, listings, 195

**iterating**

lists, 377

maps, 381-382

**IU functions, toolbar applications, 224**

**killing cells, 710-711**

## **L**

---

**learning tools, API, 8**

**left button clicks, coding, 52**

**lfHeight fields, fonts, determining, 84**

**Life**

application classes, listings, 721-730

application constants, 739

bitmaps

features, 744

starting images, 744

brushes, choosing, 746

cells

displaying, 746

requirements, 752

Command-UI Response Functions, 740

examining, 720

implementation, 710

main loops, simulations, 747

pointers, bitmaps, 739

programs, 719-720

- listings, 738

- starting, 748

- simulation, 710

- status bars, 742-744

- timers, 748

- toolbars, 742-744

- variables, initializing, 741

- windows, creating, 741

## **limiting access, CCntlDlg, 149**

### **lines**

- color, RGB macro, 86

- drawing, pens, 87

- horizontal, 321

- positioning, MoveTo(), 87

- styles, controlling, 87

- thickness, ShowPens(), 86

### **linked lists**

- contents, displaying, 717

- creating, 712

- demonstrations, 713

- features, 712

- initializing, 748

- list-handling operations, 716

- loops, 714

- nodes

- getting, 749

- listings, 712

- printing, 715

- removing, 749

- object-oriented, 715

- parameters, 717

- pointers, 713

- see also* lists

### **linking**

- applications, device drivers, 74

- controls, Windows 95, 164

- objects

- container applications, 524-526

- definition, 515

- pages, 656

## **list boxes**

- controls, 140
- initializing, 151

## **list view**

- changing, 196
- columns, creating, 192-193
- controls
  - Explorer, 185
  - features, 184
  - notifications, 196-197
- creating, 189
- image lists
  - associating, 192
  - listings, 190
- initializing, 191
- items
  - creating, 194-196
  - listings, 195
- manipulating, 196-198
- styles, 191

## **list-handling operations, linked lists, 716**

### **listings**

- <OBJECT> tags, 605
- accessors, get\_CaptionProp, 648
- ActiveX
  - constructors, 558
  - initializing, 574
- AfxOleInit(), calling, 527, 554
- applications windows, repainting, 51
- AppWizard, CApp1Doc, 40
- automation servers, codes, 555
- BITMAPFILEHEADER Structure, 326
- BITMAPINFO Structures, DIBs, 327
- BITMAPINFOHEADER Structures, DIBs, 327
- bitmaps, checking, 336
- CApp1View, pointers, 41
- Car 2 applications, 788-790
- casting procedures, 441
- catch blocks, multiple, 425-426
- CBitmapView

- constant declarations, 317
  - functions, 315-316
- CBrush Objects, creating, 88
- CDib classes, implementing, 331-335
- CDib header files, 329-330
- CDlg1 Objects, constructing, 133
- cells, processing, 751
- character arrays, constructors, 810
- CHARFORMAT, structures, 271
- check boxes, status, 153
- class derivation, 439-440
- class templates
  - defining, 430
  - multiple parameters, 433
  - parameters, 434
  - procedures, 431-432
- classes, exceptions, 421
- clipboards, support, 687
- CMainFrame
  - header files, 210-211
  - implementation files, 211-213
- CMyToolbar
  - header files, 227
  - implementation files, 227-229
- CNode structures, 374
- colors, numbers, 338
- combo boxes
  - creating, 230
  - initializing, 152, 230
- Command-UI Response Functions, Life, 740
- constants, dispid, 648
- constructors
  - CDib, 335
  - view classes, 502
- controls
  - redrawing, 643
  - updating, 655-656
- Create() prototypes, 65
- CreateListView() functions, 180-181, 186-187
- CreateProgressBar() functions, 165-166

- CreateSlider() functions, 171
- CreateSpinner() functions, 176
- CREATESTRUCT, 91
- CreateTreeView() functions, 180-181, 199-200
- CSomeResource implementation files, 414-415
- CToolBarApp
  - header files, 209
  - implementation files, 210
- data
  - initializing, 298
  - retrieving, 137
  - transferring, 136
- data members
  - CDib, 330
  - declarations, 189, 507
  - declaring, 150
  - initializing, 136
- data strings, editing, 304-305
- declarations, 538
- dialog boxes
  - data members, 131
  - storage, 135
- directories, contents, 511
- dispatch maps, document classes, 576
- display strings, changing, 308
- document data, declaring, 54
- DoDataExchange(), overriding, 133
- DoFilter( ) functions, 474
- drags, constraining, 699
- DrawGrid( ) functions, 320
- enumerations
  - alignment, 639
  - dispid, 645
- exception handling codes, 420-421
- exceptions, catching, 423
- files, reading, 309-310
- functions
  - CreateDibPalette( ), 356
  - DeleteContents( ), 348
  - OnInitDialog, 654-655

- generations, setting, 754
- GetDataFromTransfer Update, 702-703
- GetPredefinedStrings, 677
- GUID, defining, 575
- header files, 62, 124-125
  - CCountArray, 407
  - CCountArray2, 411
  - CDlg1 Class, 128
  - CMainFrameClass, 64
  - CMessages, 301-302
  - dialog box classes, 146
  - including, 132
  - main window classes, 142-143
  - menu applications, 103-104
  - MFC App2 applications, 70-71
  - point classes, 798
  - shape classes, 798
- helper functions
  - CopyDataToClipboard, 688-689
  - CopyStgMedium, 690
  - PrepareDataForTransfer, 689
- implementation files
  - CDlg1 Class, 129
  - CMainFrame, 64-65
  - CMessages, 302-303
  - dialog box classes, 147-148
  - GetDataFromClipboard, 690-691
  - GetDataFromTransfer, 691-692
  - IEnumFORMATETC, 686
  - main window classes, 143-146
  - menu applications, 104-108
  - MFC App2 applications, 71
  - OnKeyDown, 693
  - OnLButtonDown, 695
  - point classes, 798
- indicator arrays, new panes IDs, 236
- indicator IDs, creating, 234
- InitInstance( ) functions, 584
- Insert New Object Commands, responding, 528-529
- interfaces

- IDataObject, 684-685
- IDropSource, 694
- IDropTarget, 700
- left button clicks, coding, 52
- Life
  - application classes, 721-730
  - application constants, 739
  - programs, 738
- lines, horizontal, 321
- linked lists
  - demonstrations, 713
  - initializing, 748
  - nodes, 749
- list boxes, initializing, 151
- list view
  - changing, 196
  - image lists, 190
  - items, 195
- LOGFONT
  - fields, 83
  - initializing, 84
- logical palettes
  - initializing, 359
  - structures, 358
- LV\_COLUMN Structures, initializing, 193
- LV\_ITEM Structures, initializing, 195
- maps, initializing, 380
- member functions, CDib, 330
- member initialization, class constructors, 688
- member variables
  - adding, 672
  - custom data format, 701
  - initializing, 640
- menus
  - message handlers, 109
  - state variables, 108
- message maps, declarations, 189
- namespaces
  - aliases, 447-448
  - defining, 445



- definitions, 446
- unnamed, 447
- new indicator arrays, 239
- nodes
  - creating, 374
  - linked lists, 712
  - structures, 712
- OnButtonClicked( ) notification functions, 601
- OnCentered( ) functions, 265
- OnChangeItemPosition( ) functions, 533
- OnConnectMakeconnection( ) functions, 492-493,501-502
- OnCreate( ) functions, 314
- OnCreateControlBars( ) member functions, 559
- OnData, calling, 674
- OnDestroy, cleanup, 681
- OnDraw( )
  - arrays, 370
  - functions, 251, 317, 348, 357, 503
  - member functions, 531, 557
  - procedures, 664-666
- OnFileSave( ) functions, 308
- OnGetEmbeddedItem( ) member functions, 559
- OnGetExtent( ) member functions, 557
- OnHScroll( ) functions,171-172
- OnLButtonDbIcIk( ) functions, 503-504,535-536
- OnLButtonDown( ) functions, 167, 315, 369
- OnLeft( ) functions, 264
- OnMove( ) functions, 467
- OnNotify( ) functions, 188
- OnPaletteChanged( ) codes, 362
- OnPropSheet( ) functions, 252
- OnQueryNewPalette( ) codes, 363-364
- OnRButtonDown( ) functions, 505-506
- OnRecordAdd( ) functions, 467
- OnRecordDelete( ) functions, 468
- OnSetActive( ) member functions, 257
- OnSetCursor( ) functions, 536-537
- OnSetFocus( ) member functions, 530-531
- OnSetSetstring( ) functions, 565
- OnSetSetxposition( ) program lines, 572

- OnSize( ) member functions, 530
- OnSortDept( ) functions, 473
- OnSortID( ) functions, 473
- OnSortName( ) functions, 473
- OnSortRate( ) functions, 473
- OnUnderlined( ) functions, 264, 275
- Paint1 message maps, 82
- PaintDC, creating, 83
- PALETTEENTRY Structures, 358
- paragraphs, formats, 272-273
- parameters
  - alignment, 641
  - ATLCONTROLWIN.CPP, 640
- pointers, color tables, 338
- PreCreateWindow( ), overriding, 90
- PreviousDirectory( ) functions, 506
- programs
  - bunny-in-barrels, 757
  - simplifying, 799
  - stacks, 760-761
- properties, alignment, 646
- property sheets
  - displaying, 259
  - return values, 255
- radio buttons
  - setting, 223
  - status, 153
- rectangles, drawing, 216
- rectangles applications
  - OnLButtonDown( ), 59
  - OnNewDocument( ) Function, 55
- recursion
  - exponentiation, 758
  - functions, 756
- registry scripts, 635
- replacement functions, procedures, 429
- resource scripts
  - dialog boxes, 121-122
  - menu applications, 98-100
- RGBQUAD Structures, DIBs, 328

- root items, creating, 204-205
- RTTI
  - downcasting, 443-444
  - type\_info classes, 435
- scope, resolving, 446
- Serialize( ) member functions, 556
- Serialize( ) Functions, rectangles applications, 56-57
- simulations
  - Car, 796
  - stopping, 751
- sliders, responding, 175
- source codes, Paint1, 75-76
- squares
  - drawing, 323
  - locating, 322
- strings, saving, 300
- subitem columns, creating, 193
- subitems, creating, 195
- templates
  - expanding, 614-615
  - functions, 427-428
- ThreadProc( ) functions, 398
- threads, functions, 409
- time spans, calculating, 390
- tm Structure, 389
- toolbar applications, IU functions, 224
- toolbars, command IU functions, 221
- tree view
  - child items, 205
  - notifications, 206
- TV\_INSERTSTRUCT Structures, 204
- TV\_ITEM Structures, 203
- typeid operators, procedures, 436-437
- UI handlers, new panes, 239
- underlining, 272
- update handlers, custom panes, 237
- Variant types, converting, 642
- view buttons, creating, 196
- Win 95 controls functions, 187-188
- wizards, buttons, 258

## **lists**

### classes

features, 371-372

member functions, 372

cleaning, 377-378

declaring, 374-375

demo applications, 372-374

initializing, 374-375

iterating, 377

next to die cells, 750

### nodes

adding, 375

deleting, 375-376

*see also* linked lists

## **loading**

animation sequences, 285

controls, TSTCON32, 636

data classes, 26

## **locating**

source codes, ATL, 614

squares, calculating, 322

## **LOGFONT**

features, 83

fields, 83

initializing, 84

text, positioning, 85

## **logical palettes**

creating, 359

deleting, purpose, 360

features, 352

initializing, 359

messages, responding, 361-362

structures, 358

## **loops, linked lists, 714**

## **LV\_COLUMN Structures**

initializing, 193

listings, 192

## **LV\_ITEM Structures, initializing, 195**

# **M**

---

## **m\_display**

- OnPaint( ), 82
- switch statements, 82-83
- values, changing, 89

## **m\_IReadyState member variables, initializing, 669**

### **macros**

- categories, 479
- IMPLEMENT\_SERIAL( ), 303
- message maps, 110, 485
- predefined, 479
- runtime services, 485-486

## **main loops simulations, Life, 747**

### **main window classes**

- header files, 142-143
- implementation files, 143-146

## **management information, 480**

### **managing**

- ActiveX control bars, 559
- DCs, CPaintDC, 82

### **manipulating**

- animations, 284
- controls
  - purpose, 149
  - Rich Edit, 268-270
- list views, 196-198
- objects, CRect, 392
- progress bars, 170
- sliders, 175
- tree view, 205-206

### **manually**

- compiling programs, 66-68
- creating applications, 62
- declaring message maps, 68
- defining message maps, 68-69
- registering controls, 635
- responding, messages, 68

## **mapping colors, windows, 353**

### **maps**

- classes
  - features, 378

- functions, 379
- creating, 380
- demo applications
  - features, 379-380
  - OnLButtonDown( ), 381
- header files, 625
- initializing, 380
- iterating, 381-382
- objects, initializing, 380
- property, 625
- values, retrieving, 380-381

## **MDI (Multiple-Document Interface)**

- child windows, 16-17
- features, 15-16

### **member functions**

- accessing, pointers, 150
- AddString(), purpose, 151
- arrays, classes, 366
- CActiveXCont1, CntrlItem classes, 527-528
- calling, GetCurSel( ), 154
- CAnimateCtrl, 285
- CBase, Func2( ), 440
- CBitmap, 319
- CDialog, 19
- CDib
  - listings, 330
  - public, 331
  - types, 337-338
- CFile classes, 307
- CFrameWnd, 14
- CFtpConnection, 509
- CHttpFile, 497
- CImageList Class, 183-184
- CInternetSession, 495-496
- classes, lists, 372
- CMDIChildWnd, 17
- CMDIFrameWnd, 16
- CObject
  - overview, 9
  - types, 10

- controls, 149
- CProgressCtrl, 169
- CPropertyPage, 22
- CPropertySheet, 21-22
- CRect, 392
- CSliderCtrl, 173-174
- CSpinButtonCtrl, 178
- CString, 385-386
- CTime, 387
- CTimeSpan, 387
- CView, 15
- CWinApp, 12
- GetDlgItem( ), procedures, 149
- GetLBText( ), 154
- OnCreateControlBars( ), 559
- OnDraw( ), 531, 557
- OnGetEmbeddedItem( ), 559
- OnGetExtent( ), 557
- OnLButtonDblClk( ), exploring, 543
- OnSetActive( ), 257
- OnSetFocus( ), 530-531
- OnSize( ), 530
- Serialize( ), 556
- SetCheck( ), 151
- see also* functions

**member initialization, class constructors, 688**

**member variables**

- adding
  - listings, 672
  - Object Wizard, 649
- alignment, 645
- custom data format, 701
- initializing
  - class constructors, 701-702
  - constructors, 662
  - listings, 640
  - m\_IReadyState, 669

**members, CREATESTRUCT, 91**

**memory, reading, DIBs, 336**

**memory allocation, successful, 421**

## **memory DCs**

DrawSquare( ), 322

OnDraw( ), 321

## **menu applications**

commands, enabling, 101

features, 101

header files, 103-104

implementation files, 104-108

resource scripts, 98-100

source codes, 102

tricks, 102

## **menu editors, default menu bars, modifying, 45**

### **menus**

commands

appearances, 113

check marks, 114

responding, 112

updating, 114

enabling, responses, 112

features, 93

IDs

defining, 96

predefined, 96-97

items, defining, 97

message handlers

features, 109

listings, 109

writing, 112-113

resources, creating, 94-96

state variables

features, 108

listings, 108

support, adding, 93-94

### **message handlers**

menus

features, 109

listings, 109

message maps, types, 110

property sheets, adding, 653-654

writing menus, 112-113



## **message maps**

- CActiveXCont1, document classes, 532
- declarations, 189
- declaring, manually, 68
- defining, manually, 68-69
- functions, writing, 70
- macros, 110, 485
- message handlers, 110
- ON\_WM\_PAINT( ), 81
- OnPaint( ), 81
- Paint1, 82

## **message response functions, creating, toolbars, 215-217**

## **message-mapping, classes, MFC, 11-12**

## **messages**

- default, file demo applications, 296
- generating, WM\_PAINT, 90
- responding
  - logical palettes, 361-362
  - manually, 68

## **methods**

- adding, controls, 602-603, 637
- controls
  - dual-interface, 622
  - testing, 603-604
- creating, ATL, 637
- CStatusBars, 231
- events, defining, 658
- OnKeyDown, clipboards, 682
- servers, automation, 576-579

## **MFC (Microsoft Foundation Classes)**

- application objects, deriving, 10-11
- applications, running, 36-37
- classes
  - categories, 9
  - features, 8-9
  - message-mapping, 11-12
  - types, 29
- comparing, ATL, 618-619
- custom window classes, purpose, 8
- DAO, 27

- dialog boxes, 18
- drawing-object classes, 24
- ODBC, 27
- purpose, 8
- Windows, comparing, 8
- windows classes
  - features, 12
  - types, 12-13

## **MFC App2 applications**

- compiling, 71
- header files, 70-71
- implementation files, 71

## **MFCRichEdit, AppWizard, procedures, 273-276**

## **mice, clicking, OnLButtonDown( ), 322**

- objects, selecting, 533

## **Microsoft Foundation Classes, *see* MFC**

## **Miscellaneous tabs, controls, 633**

## **modifying**

- appearances, panels, 237-238
- data members, view objects, 41
- default menu bars, menu editors, 45
- document classes, procedures, 48-50
- document data, view objects, 58-59
- resources, ShowDib, 341-345
- view classes, procedures, 51-52

## **monitors, colors, displaying, 74**

## **mouse, pointer notifications, 706**

## **MoveTo( ), lines, positioning, 87**

## **moving data, dialog boxes, 137**

## **multiple**

- catch blocks
  - exceptions, 424
  - procedures, 425-426
- parameters, class templates, 433

## **multiple-document interface, *see* MDI**

## **mutexes**

- definition, 411
- objects, accessing, 412
- releasing, 412

# **N**

---

## **namespaces**

- aliases

  - features, 447

  - procedures, 447-448

- defining, 445

- nesting, 446

- scopes, defining, 444-445

- shortcomings, 444-445

- unnamed, defining, 447

- user-defined, accommodating, 445

## **naming interfaces, conventions, 625**

## **neighbor counts, incrementing, cells, 750**

## **nesting, namespaces, definition, 446**

## **new applications, ActiveX, 518**

## **new indicator arrays, 239**

## **new panes, UI handlers, 239**

## **new panes IDs, indicator arrays, adding, 236**

## **next to die lists, cells, adding, 750**

## **nodes**

- adding, lists, 375

- creating, 374

- deleting, lists, 375-376

- getting, linked lists, 749

- implementing, trees, 779

- linked lists

  - listings, 712

  - printing, 715

- removing, linked lists, 749

- structures, 712

## **notification functions**

- buttons, adding, 600-601

- OnButtonClicked( ), 601

- properties, 600

## **notification methods, properties, creating, 595**

## **notifications**

- list view, controls, 196-197

- tree view, 206

## **numbers, colors, 338**

# **O**

---

**Object Linking and Embedding, *see* OLE**

**Object-Oriented Programming, *see* OOP**

**Object Wizard**

codes, generating, 624

member variables, adding, 649

**Object Wizard dialog boxes, ATL, 616**

**Object Wizards, ATL, procedures, 620**

**object-oriented, linked lists, 715**

**objects**

accessing, mutexes, 412

adding, CPoint, 42

comparing, 43

configuring, CFile, 26-27

creating

CArchive, 310

CDib, 335

CFtpConnection, 509

CHttpFile, 495-496

CInternetSession, 495

CPoint, 390

CRect, 391

CSize, 391

CString, 481

CTime, procedures, 388

CTimeSpan, 390

data

initializing, 304

serializing, 305

data members, encapsulating, 790

defining

classes, 802

OOP, 790

deleting

ActiveX, 547-549

CPaintDC, 82

document, 39

embedding

container applications, 523-524

definition, 516

exceptions, types, 422

- instantiating, class templates, 430
- linking
  - container applications, 524-526
  - definition, 515
- manipulating, CRect, 392
- maps, initializing, 380
- persistent, creating, 300
- properties
  - miscellaneous, 622
  - stock, 623-624
- property sheets, adding, 652-653
- selecting
  - ActiveX, 539-540
  - mice, 533
- sharing, OLE 2, 516
- strings, concatenation operators, 811
- uninitialized
  - CPoint, 390
  - CRect, 392
  - CSize, 391
- view, 39
- see also* drawing objects

**obstacles, finding, encapsulating, 791**

**obtaining pointers, CApp1View, 41-42**

**OCX, 630**

**ODBC (Open Database Connectivity)**

- classes, 449

- DAO

- comparing, 477-478

- differences, 478

- similarities, 478

- databases, creating, 453

- features, 27

**OK buttons, dialog boxes, 152-155**

**OLE (Object Linking and Embedding)**

- automation

- definition, 517

- features, 569

- initializing, 576

- comparing

- ActiveX, 518
- COM, 518
- definition, 515
- events, responding, 657
- servers
  - displaying, 575
  - initializing, 574-575
  - registering, 575

## **OLE 2**

- functionality, 516
- objects, sharing, 516

**ON\_WM\_PAINT( ) message maps, 81**

**OnButtonClicked( ) notification functions, 601**

**OnCentered( ) functions, 265**

**OnChangeItemPosition( ) functions, 533**

**OnConnectMakeconnection( ) functions, 492-493, 501-502**

### **OnCreate**

- functions
  - bitmap applications, 319
  - listings, 314
- implementations, 697

**OnCreateControlBars( ) member functions, 559**

**OnData, calling, 674**

### **OnDestroy**

- cleanup, 681
- implementations, 698-699

**OnDialogTest() dialog boxes, displaying, 136**

### **OnDraw**

- data, displaying, 57
- functions
  - arrays, 370
  - exploring, 544-545
  - listings, 251, 317, 348, 357, 503
  - ShowDib, 350-351
- member functions, 531, 557
- memory DCs, creating, 321
- parameters, single, 58
- procedures, 664-666
- rectangles applications, 57-58
- ShowDib, changing, 359-360

- OnEditClear( ) functions, exploring, 550**
- OnFileOpen( )**
  - codes, 347
  - procedures, ShowDib, 349
- OnFileSave( ) functions, 308**
- OnGenerations( ) functions, File, 754**
- OnGetEmbeddedItem( ) member functions, 559**
- OnGetExtent( ) member functions, 557**
- OnHScroll( ) functions, 171-172**
- OnInitDialog functions, 654-655**
- OnInitDialog( ) base class versions, calling, 150**
- OnKeyDown**
  - implementations, 693
  - methods, clipboard, 682
- OnLButtonDblClk( )**
  - functions, 503-504, 535-536
  - member functions, exploring, 543
- OnLButtonDown implementations, 695**
- OnLButtonDown( )**
  - functions, 89
    - exploring, 542-543
    - listings, 167, 315, 369
  - mice, clicking, 322
  - parameters, 59-60
  - rectangles applications, 59
- OnLeft( ) functions, 264**
- OnMove( ) functions**
  - examining, 469
  - listings, 467
- OnNewDocument( ) functions**
  - calling, 55-56
  - rectangles applications, 55
- OnNotify( ) functions, 188**
- OnPaint( )**
  - CPaintDC objects, deleting, 82
  - m\_display, 82
  - message maps, 81
  - windows, displaying, 81
- OnPaletteChanged( ) codes, 362**
- OnPaletteChanged() functions, examining, 363-364**

- OnPropSheet() functions, 252**
- OnQueryNewPalette( ) codes, 363**
- OnQueryNewPalette() functions, examining, 364**
- OnRandomCells() functions, File, 753**
- OnRButtonDown() functions, 505-506**
- OnRecordAdd() functions**
  - examining, 468
  - listings, 467
- OnRecordDelete() functions**
  - examining, 470
  - listings, 468
- OnSetActive() member functions, 257**
- OnSetCursor() functions**
  - exploring, 543-544
  - listings, 536-537
- OnSetFocus() member functions, 530-531**
- OnSetSetstring() functions, 565**
- OnSetSetxposition(), program lines, 572**
- OnSize() member functions, 530**
- OnSortDept() functions**
  - examining, 475
  - listings, 473
- OnSortID() functions, 473**
- OnSortName() functions, 473**
- OnSortRate() functions, 473**
- OnSpeed() functions, File, 754**
- OnUnderlined() functions, 264, 275**
- OOP (Object-Oriented Programming)**
  - classes
    - constructors, 798
    - data types, 791
    - general, 797
    - problems, 797
  - CMDIChildWnd classes, functionality, 17
  - downcasting, definition, 435
  - encapsulation, definition, 786
  - header files, 791
  - implementation files, 791
  - inheritance, definition, 792
  - objects, defining, 790



- power, 801
- reviewing, 785
- variables, handling, 790

**Open Database Connectivity, *see* ODBC**

**opening files, CFile, 310**

**operations, list-handling, linked lists, 716**

**operators**

- arrays, defining, 804
- assignment, character arrays, 811
- classes, CPoint, 391
- concatenation
  - character arrays, 812
  - string objects, 811
- CRect, 393
- CString, 386
- overloading
  - definition, 802
  - logically, 803-804
- performance, 805
- strings
  - assignments, 810
  - comparing, 812-813

**optimized drawing**

- capabilities, ATL, 679
- codes, 706
- drawing objects, creating, 679
- drawing resources, caching, 680

**optional parameters, VARIANT, 638**

**options**

- exceptions, handling, 426
- property sheets, demo applications, 243
- status bars, 233

**ordering**

- catch blocks, 424
- program blocks, 424

**output, Car 1 applications, 788**

**overcoming, scope-resolution problems, 445**

**overloading**

- cautions, 805
- default arguments, differences, 802

- definition, 802
- differences, 802
- operators, logically, 803-804

### **overriding**

- DeleteContents( ), CDocument, 348
- DoDataExchange( ), 133
- PreCreateWindow( ), 90
- view classes, 43
- virtual functions, 12

### **ownership, releasing, critical sections, 407**

## **P-Q**

---

### **pages**

- controls, linking, 656
- wizards, creating, 257

### **Paint1**

- fonts, creating, 85
- message maps, 82
- pens, 86
- source codes
  - directories, 76
  - listings, 75-76

### **PaintDC, creating, 83**

### **PALETTEENTRY Structures, 358**

### **palettes, handling, ShowDib, 355**

### **panels**

- appearances, modifying, 237-238
- status bars, styles, 238

### **paragraphs**

- aligning, Rich Edit, 272
- formats, changing, 272-273

### **parameterized properties, 647**

### **parameters**

- <OBJECT> tags, 605
- alignment
  - listings, 641
  - ShowCaption, 641
- ATLCONTROLWIN.CPP, 640
- attributes
  - IDL, 638

- purpose, 638
- catch programs, 421
- CDlg1, 133
- data types, class templates, 434
- entering, exceptions, 424
- flowing
  - attributes, 638
  - direction, 638
- FTP servers, default values, 512
- functions, recursion, 760
- linked lists, 717
- OnDraw( ), single, 58
- OnLButtonDown( ), 59-60
- optional, VARIANT, 638
- strings
  - comparing, 813
  - concatenation, 812
- update-command-UI functions, 113
- parsing, recursion, 759**
- parts, status bars, 231**
- pens**
  - features, 86
  - lines, drawing, 87
  - styles, 87
- performance operators, 805**
- persistence**
  - adding, 674-675
  - advantages, 661
  - properties, adding, 661
- persistent objects, creating, 300**
- persistent classes, creating, 299**
- placing**
  - codes, exception-handling, 420
  - data, clipboards, 687
  - squares, DrawSquare( ), 322
- point arrays, points, adding, 52**
- point classes**
  - header files, 798
  - implementation files, 798
- pointer notifications, mouse, 706**

## **pointers**

- bitmaps, Life, 739
- color tables, 338
- creating
  - CBase, 442
  - polymorphism, 795
- data types, correct, 337
- dynamic\_cast downcasting, 442
- linked lists, 713
- member functions, accessing, 150
- necessity, polymorphism, 796
- obtaining, CApp1View, 41-42
- variables, accessing, 43

## **points, adding, point arrays, 52**

### **polymorphism**

- failed, 442
- features, 440
- ignoring, 442-444
- pointers
  - creating, 795
  - necessity, 796
- purpose, 794
- rectangles, drawing, 800

### **Pos( ) functions**

- character arrays, 814
- strings, 813

### **positioning**

- lines, MoveTo( ), 87
- text, LOGFONT, 85
- windows, 90

### **power, OOP, 801**

### **Power( ) exponentiation, recursion, 758**

### **PreCreateWindow( )**

- calling, base classes, 91
- overriding, 90

### **predefined**

- constants, 479
- IDs, menus, 96-97
- macros, 479
- variables, 479

**PrepareDataForTransfer** helper functions, 689

**PreviousDirectory( )** functions, 506

**printing nodes, linked lists**, 715

**priorities, constants, threads**, 396

**problems, function templates**, 428

**processing**

cells, 751

documents, HTML, 497

exceptions, 484

**processors, speed, advantages**, 2

**program blocks, ordering**, 424

**program lines, OnSetSetxposition( )**, 572

**programming**

applications

FTP, 498-501

Web, 488-492

computers, introduction, 7

Windows, API, 8

*see also* OOP

**programs**

application objects, instantiating, 63

bunny-in-barrels, 757

compiling, manually, 66-68

creating

approaches, 31

examples, 32-36

manually, 61

Life, 719-720, 738

simplifying, 799

stacks, 760-761

starting, Life, 748

**progress bars**

adding, 165-168

creating, 169

initializing, 169-170

manipulating, 170

procedures, 168

**projects**

ATL

types, 618

creating, 619

## **properties**

ActiveX controls, categories, 643-644

adding

asynchronous, 667-668

controls, 594

persistence, 661

alignment, 646

ambient, 651

ButtonCaption, 596

COM, setting, 621

controls

testing, 597-598

values, 597

creating

Get( ) property methods, 595

notification methods, 595

Set( ) property methods, 595

data, encapsulating, 643

dialog boxes, ATL Object Wizard Properties, 632

dispid, BackColor, 650

fonts, ambient, 663

notification functions

buttons, 600-601

features, 600

objects

miscellaneous, 622

stock, 623-624

parameterized, 647

persistence, adding, 674-675

persistent, 599

ReadyState

adding, 668-669

implementing, 669

servers, automation, 576-579

stock

adding, 649

features, 649

TextDataPath, 672

updating, controls, 655

user-defined, creating, 644-645

**Properties dialog boxes, check boxes, 622**

**property enumerations**

dynamic, 675-676

features, 675

**property maps, 625**

**property pages**

classes, creating, 248-249

entries, adding, 656

resources, 245-247

selecting, 676

**property sheets**

classes, creating, 248-249

creating, 20

definition, 241

demo applications

configuring, 249-252

creating, 243-245

features, 253-255

options, 243

running, 252-253

source codes, 243

displaying, 259

features, 20, 651

finding, Windows 95, 241

message handlers, adding, 653-654

objects, adding, 652-653

return values, 255

*see also* dialog boxes; wizards

**protecting data members, declaring, 132**

**prototypes, Create( ), 65**

**proxy codes, generating, 658-659**

**public**

dialog boxes, data members, 131

member functions, CDib, 331

**public data members**

CFrameWnd, types, 13

CWinApp, 11-12

**Pushbuttons, controls, 140**

**QueryContinueDrag, purpose, 696**

# R

---

## **radio buttons**

- controls, 140
- creating toolbars, 222-224
- default states, 151
- initializing, CControlDlg, 150
- setting, 223
- status, storing, 153

## **reading**

- arrays, 370-371
- documents, HTML, 496-497
- files
  - directly, 305
  - listings, 309-310
- memory, DIBs, 336
- root directories, FTP servers, 509-510

## **ReadyState properties**

- adding, 668-669
- implementing, 669

## **ReadyStateChange events, adding, 670**

## **receiving references, Serialize( ) functions, 57**

## **records, databases**

- adding, 463-468
- deleting, 463-468
- filtering, 470-473
- sorting, 470-473

## **Rectangle( ) functions, procedures, 88**

## **rectangles**

- creating, new, 801
- drawing
  - listings, 216
  - polymorphism, 800

## **rectangles applications**

- creating, AppWizard, 43-44
- data, storing, 54
- exploring, 53
- OnDraw( ), 57-58
- OnLButtonDown( ), 59
- OnNewDocument( ) Function, 55
- running, 53



saving, 53

Serialize( ) Functions, 56-57

### **rectangle classes**

header files, 799

implementation files, 799

### **recursion**

applications, examples, 762-777

definition, 755-756

exponentiation

listings, 758

Power( ), 758

frames, stacks, 760

functions

conditional statements, 761

DoBlanks( ), 781

examples, 756-757

listings, 756

parameters, 760

parsing, 759

routines, caveats, 762

sorting, 759

tree-traversal, 759

trees, traversing, 777

### **redrawing, controls, 643**

### **references, receiving, Serialize( ) functions, 57**

### **refreshing, containers, FireOnChanged, 646**

### **registering**

ActiveX, server applications, 555

controls

ATL, 634

manually, 635

databases, procedures, 453-455

server applications, ActiveX, 552

servers, OLE, 575

targets, procedures, 697

### **registry scripts, 635**

### **relational databases**

accessing, 452

definition, 450-452

### **releasing**

- mutexes, 412

- ownership, critical sections, 407

## **removing**

- dialog boxes, procedures, 154

- elements, arrays, 371

- nodes, linked lists, 749

## **repainting**

- applications windows, 51

- Invalidate( ), 60

## **replacement functions**

- procedures, 429

- templates, 429

## **representing strings**

- character arrays, 806

- dynamically allocated memories, 806

## **requirements**

- clipboards, interfaces, 683

- drags, initiating, 694

- Internet Explorer, controls, 632

- Life, cells, 752

## **resolving scope, 446**

## **resource files**

- constants, 97-98

- features, 97

## **resource access counting, semaphores, 413**

## **resource editors, dialog boxes, creating, 18-20**

## **resource scripts**

- dialog boxes, 121

- features, 98

- menu applications, 98-100

- writing, 100

## **resources**

- creating

  - dialog boxes, 119-120

  - menus, 94-96

  - property pages, 245-247

  - toolbars, 214-215

- modifying, ShowDib, 341-345

## **ResourceView windows, displaying, 44**

## **responding**

- buttons

  - toolbars, 216

  - wizards, 258

- commands

  - IDs, 113

  - Insert New Object, 528-529

  - menus, 112

- events, OLE, 657

- menus, enable, 112

- messages

  - logical palettes, 361-362

  - manually, 68

- sliders, 175

- responsibilities, document objects, 54**

- retrieving**

  - data, 137

  - strings, 816

  - values, maps, 380-381

- return values**

  - GetDlgItem( ), 150

  - property sheets, 255

- returning data, callers, 705**

- reversing, animate applications, 285**

- reviewing**

  - DDBs, 325

  - DIBs, 325

  - OOP, 785

- RGB macro, lines, color, 86**

- RGBQUAD Structures, DIBs, 328**

- Rich Edit**

  - applications

    - compiling, 267, 275

    - creating, 262-267

  - controls

    - adding, 273-275

    - creating, 268

    - initializing, 268-270

    - manipulating, 268-270

    - features, 261-262

    - overview, 261

- paragraphs, aligning, 272
- settings, AppWizard, 263-264
- toolbars, 270
- root directories, reading, FTP servers, 509-510**
- root items, creating, 204-205**
- routines, recursion, caveats, 762**
- RTTI (Run-Time Type Information)**
  - definition, 419
  - downcasting, 443-444
  - enabling, 437-438
  - purpose, 434, 439
  - type\_info classes, 435
- rules, simulations, Conway, John, 709**
- Run-Time Type Information, *see* RTTI**
- running**
  - ActiveX, server applications, 566
  - applications
    - MFC, 36-37
    - threads, 410
  - basic calculator, ActiveX, 590-591
  - client applications, automation, 585
  - console applications, procedures, 419
  - demo applications
    - property sheets, 252-253
    - wizards, 255-256
  - documents, CMDIFrameWnd, 16
  - rectangles applications, 53
  - ShowDib
    - procedures, 351-352
    - version 2, 360-361
- runtime services, macros, 485-486**

## **S**

---

### **saving**

- data, classes, 26
- procedures, << and >> operators, 57
- rectangles applications, 53
- strings, 300
- scope-resolution problems, overcoming, 445**
- scopes**

- defining, namespaces, 444-445
- resolving, 446
- scripts, registry, 635**
- scroll bars, controls, 140**
- SDI (Single Document Interface), applications**
  - features, 297
  - window classes, 14
- searching strings, purpose, 813**
- selecting**
  - brushes, SelectObject( ), 88
  - objects
    - ActiveX, 539-540
    - mice, 533
  - property pages, 676
- SelectObject( )**
  - brushes, selecting, 88
  - functions, bitmaps, 320
- semaphores**
  - creating, 413
  - definition, 413
  - header files, CSomeResource, 414
  - resource access, counting, 413
- sending**
  - data, custom formats, 704
  - WM\_PAINT messages, reasons, 81
- sequences, animations**
  - finding, 277
  - loading, 285
  - Windows 95, 277
- Serialize( )**
  - functions
    - document classes, 299
    - ShowDib, 349
    - rectangles applications, 56-57
    - references, receiving, 57
  - member functions, 556
- serializing data**
  - CObject, 10
  - document, 56-57
  - objects, 305

## **server applications, ActiveX**

- compiling, 566
- creating, 551-554
- definition, 520
- functionality, 554
- registering, 552, 555
- running, 566
- testing, 553

## **servers**

- automation
  - creating, 569-570
  - methods, 576-579
  - properties, 576-579
- OLE
  - displaying, 575
  - initializing, 574-575
  - registering, 575

## **services, CWinApp, 10**

**Set() property methods, properties, creating, 595**

**SetButtonInfo() functions, arguments, 229**

**SetCheck( )member functions, purpose, 151**

**SetSelectedItem() functions, exploring, 546-547**

## **setting**

- alignment functions, 646
- buttons, wizards, 257
- colors
  - applications, 353
  - ShowDib, 352
- properties, COM, 621
- radio buttons, 223
- sliders, values, 174
- styles, toolbars, 218-219
- values
  - COLORREF, 88
  - generations, 754

**settings, AppWizard, RichEdit, 263-264**

## **shape classes**

- header files, 798
- test programs, 799

**sharing objects, OLE 2, 516**

**ShowBrushes( ) functions, 88**

**ShowCaption**

calling methods, 640

parameters, alignment, 641

**ShowDib**

applications, creating, 338-341

codes, adding, 345-346

colors, setting, 352

OnDraw( )

changing, 359-360

functions, 350-351

OnFileOpen( ), procedures, 349

palettes, handling, 355

resources, modifying, 341-345

running, procedures, 351-352

Serialize( ) functions, 349

version 2, running, 360-361

version 3

codes, 362

creating, 361

**showing toolbars, 221-222**

**ShowPens( ), line thickness, 86**

**similarities, DAO, ODBC, 478**

**simplifying programs, 799**

**simulation, Life, 710**

**simulations**

Car, listings, 796

main loops, Life, 747

rules, Conway, John, 709

stopping, 751

**single parameters, OnDraw(), 58**

**Single Document Interface, *see* SDI**

**single-instance classes**

appropriateness, 802

problems, 802

**sizes**

changing, stacks, 761

controlling, BITMAPINFOHEADER structures, 327

determining, image data, 328

windows, overview, 90

## **sliders**

- configuring, 174
- creating, 171-173
- features, 170-178
- initializing, 173
- manipulating, 175
- responding, 175
- styles, 173
- values, setting, 174

## **sorting**

- records, databases, 470-473
- recursion, 759

## **source code files, CCntIDlg, 146**

## **source codes**

- animate applications, 278
- applications, autoserver, 570
- Car3 applications, 794
- character arrays, constructors, 810
- client applications, automation, 580
- container applications, ActiveX, 521-522
- file demo applications, 297
- InitInstance( ), 554
- locating, ATL, 614
- menu applications, 102
- Paint 1, directories, 76
- property sheets, demo applications, 243

## **source strings, 811**

## **sources, enabling, Drag-and-Drop, 694**

## **spaces, allocating dynamically, 806**

## **specialized classes, CView, 15**

## **specifying alignment settings, enumerations, 638**

## **speed**

- cells, 711
- processors, advantages, 2

## **spinners**

- configuring, 178
- creating, 177-178
- features, 175
- initializing, 177

## **SQL, databases, 452**



## **squares**

- drawing, 323
- locating, calculating, 322
- placing, DrawSquare( ), 322

## **stacks**

- frames, definition, 760
- programs, 760-761
- sizes, changing, 761

## **stand-alone applications, autoserver, 573**

## **standard IDs, windows, 486**

## **starting**

- file demo 3 applications, 307
- programs, Life, 748

## **starting images, bitmaps, Life, 744**

## **state variables, menus**

- features, 108
- listings, 108

## **Static text, 140**

## **status, storing, check boxes, 153**

## **status bars**

- custom panes, adding, 235
- features, 231
- indicator arrays, creating, 234
- Life, 742-744
- options, 233
- parts, 231

## **Stock Properties**

- adding, 649
- features, 649
- objects, 623-624
- tabs, controls, 633

## **stopping simulations, 751**

## **storage**

- classes, string, 806
- dialog boxes, 135
- document classes, customizing, 54

## **storing**

- data
  - CDocument, 14-15
  - rectangles applications, 54

- status

- check boxes, 153

- radio buttons, 153

## **StretchDIBits( ) functions, 351**

### **string classes**

- description, 808

- features, 805

- functions, 807

- header files, 807

- storage, 806

### **string literals, 676**

### **string values, enumerations, getting, 676**

### **strings**

- assignments, operators, 810

- classes, testing, 816

- comparing

- features, 812

- parameters, 813

- concatenation

- definition, 811

- parameters, 812

- constructing, 810

- deleting, 815

- destructing, 810

- enumerations, changing, 678-679

- extracting, 815-816

- Format( ), 388

- handling, 806

- inserting, 814-815

- operators, comparing, 812-813

- representing

- character arrays, 806

- dynamically allocated memories, 806

- retrieving, 816

- saving, 300

- searching, purpose, 813

- source, 811

### **structured-programming techniques, 785**

### **structures**

- CHARFORMAT, 271

- DIBs, types, 326
- logical palettes, 358
- nodes, 712

## **styles**

- controlling, lines, 87
- defining, CPen constructors, 87
- list view, 191
- panels, status bars, 238
- pens
  - setting, 87
  - toolbars, 218-219
- sliders, 173
- toolbars, types, 218
- tree view, controls, 202

## **subitems, creating**

- columns, 193
- listings, 195

## **SubNbrs() functions, File, 753**

## **successful memory allocation, 421**

## **summaries, attributes, controls, 632-633**

## **support**

- adding, menus, 93-94
- clipboard
  - features, 681
  - listings, 687
- clipboards, types, 683

## **supporting Drag-and-Drop, overview, 693**

## **switch statements, m\_display, features, 82-83**

## **synchronization, threads, 406**

## **system palettes, 352**

# **T**

---

## **tabs, controls**

- Miscellaneous, 633
- Stock Properties, 633

## **targets**

- enabling, Drag-and-Drop, 696
- registering, procedures, 697

## **templates**

- classes, CList, 615

- creating
  - dialog boxes, 130
  - InitInstance(), 554
- expanding, 614-615
- functions
  - creating, 427
  - defining, 427
  - listings, 427-428
- overview, 427
- replacement functions, providing, 429
- see also* class templates; function templates

## **terminology, ActiveX, 518**

## **test programs, shape classes, 799**

## **testing**

- ActiveX server applications, 553
- buttons, controls, 601-602
- classes, strings, 816
- controls
  - properties, 597-598
  - TSTCON32, 636
- methods, controls, 603-604

## **text**

- drawing, fonts, 85
- positioning, LOGFONT, 85

## **TextDataPath**

- accessors, transferring, 673
- properties, 672

## **TextOut( ) arguments, 86**

## **ThreadProc( ) functions, 398**

## **threads**

- applications
  - building, 396-398
  - CCountArray2, 413
  - compiling, 410
  - running, 410
- communicating
  - event objects, 403-406
  - global variables, 399-401
  - user-defined messages, 401-403
- critical sections

- creating, 406
  - features, 406
- definition, 11, 395
- features, Windows 95, 395
- functions, 409
- priorities
  - constants, 396
  - definition, 396
- synchronization, 406
- throwing functions, exceptions, 424**
- time spans, calculating, 390**
- timers, Life, 748**
- tm Structure, 389**
- toolbar applications**
  - features, 208
  - final versions, creating, 238
  - IU functions, 224
- toolbars**
  - buttons, responding, 216
  - combo boxes, adding, 229
  - command IU functions, 221
  - creating, 217-218
    - AppWizard, 207
    - custom, 224-226
  - displaying, 217-218
  - dragging, 208
  - features, 207
  - hiding, 221-222
  - Life, 742-744
  - message response functions, creating, 215-217
  - radio buttons, creating, 222-224
  - resources, creating, 214-215
  - Rich Edit, 270
  - showing, 221-222
  - styles
    - setting, 218-219
    - types, 218
- tools, API, learning, 8**
- transferring**
  - accessors, TextDataPath, 673

- data

- DoDataExchange( ), 134

- get\_TextDataPath, 672-673

- listings, 136

- formats, customizing, 701-702

## **transparent styles, animations, controls, 291**

### **tree view**

- controls

- child items, 205

- features, 198

- styles, 202

- creating, 201-202

- flags, types, 203

- initializing, 202

- items, creating, 203-205

- manipulating, 205-206

- notifications, 206

### **tree-traversal**

- examples, 780

- functions, 780-781

- procedures, 779-780

- recursion, 777

### **trees**

- binary

- creating, 778-779

- definition, 777

- definition, 777

- nodes, implementing, 779

- traversing

- examples, 780

- functions, 780-781

- procedures, 779-780

- recursion, 777

## **tricks, menu applications, 102**

### **TSTCON32, controls, 636**

### **TV\_INSERTSTRUCT Structures, 204**

### **TV\_ITEM Structures, 203**

### **type\_info classes, 435**

### **typeid operators**

- information, obtaining, 435

procedures, 436-437  
types, file mode flags, 309

## U

---

**UI handlers, new panes, 239**

**unclipped device context, 706**

**underlining, 272**

**uninitialized objects**

CPoint, 390

CRect, 392

CSize, 391

**unnamed namespaces, defining, 447**

**update handlers**

custom panes, 237

*see also* command update handlers

**update-command-UI functions**

declaring, 109-110

defining, 111

parameters, 113

writing, 113-114

**updating**

bitmaps, areas, 323

commands

appearances, 113

menus, 114

controls

listings, 655-656

properties, 655

windows, bitmaps, 319

**user interfaces**

compiling, 593

creating, 591-594

Windows 95, 24

**user-defined**

namespaces

accommodating, 445

properties, creating, 644-645

messages, communicating, 401-403

**utilities classes, overview, 385**

# V

---

## values

- calculating, BITMAPINFOHEADER structures, 328
- changing, m\_display, 89
- controls, properties, 597
- enumerations, getting, 678
- retrieving, maps, 380-381
- setting
  - COLORREF, 88
  - generations, 754
- sliders, 174

## variables

- cached, cleaning, 680
- document data, 54
- handling
  - global, 790
  - OOP, 790
- initializing
  - Life, 741
  - Variant, 642
- pointers, accessing, 43
- predefined, 479

## VARIANT

- parameters, optional, 638
- variables, initializing, 642
- types, converting, 642

## version 2, ShowDib, running, 360-361

## version 3, ShowDib

- codes, 362
- creating, 361

## view buttons, creating, 196

## view classes

- ActiveX, 558
- CActiveXCont1, 528
- constructors, 502
- modifying, procedures, 51-52
- responsibilities, 41
- virtual functions, overriding, 43

## view objects

- data members, displaying, 41



document data, modifying, 58-59

features, 39

**view windows, CView, functionality, 14**

**virtual functions**

classes, creating, 805

definition, 805

implementing, 797

overriding, 12

view classes, 43

**virtual member functions, CApp1Doc, types, 41**

## **W-X-Y-Z**

---

**warnings, data loss, 60**

**Web applications**

compiling, 493

programming, 488-492

**Web pages, controls, ActiveX, 604**

**window classes, deriving, CWnd base, 13**

**windowless controls, activating, 705-706**

**Windows**

API functions, 359

clicking, applications, 75

colors, mapping, 353

controls

history, 159

types, 139-140

creating, Life, 741

deriving, SDI applications, 14

displaying

dialog boxes, 154

OnPaint( ), 81

ResourceView, 44

frames, 13

IDs, standard, 486

MFC, comparing, 8

objects, comparing, 43

positioning, overview, 90

programming, API, 8

sizing, overview, 90

updating bitmaps, 319

## **Windows 95**

- animation sequences, 277
- controls
  - compiling, 164
  - creating, 159-164
  - linking, 164
  - listings, 187-188
- property sheets, finding, 241
- threads, 395
- user interfaces, 24

## **windows classes, MFC**

- features, 12
- types, 12-13
- see also* classes

## **WinInet**

- classes, types, 487-488
- definition, 487

## **wizards**

- buttons
  - listings, 258
  - responding, 258
  - setting, 257
- class member functions, purpose, 22
- demo applications, running, 255-256
- displaying, procedures, 259
- overview, 31-32
- pages, creating, 257
- procedures, examples, 32-36
- see also* property sheets

## **WM\_PAINT messages**

- client areas, invalidating, 81
- generating, 90
- sending, reasons, 81

## **writing**

- classes, dialog boxes, 130-131
- files, directly, 305
- functions, message maps, 70
- message handlers, menus, 112-113
- resource scripts, 100
- update-command-UI functions, 113-114

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.