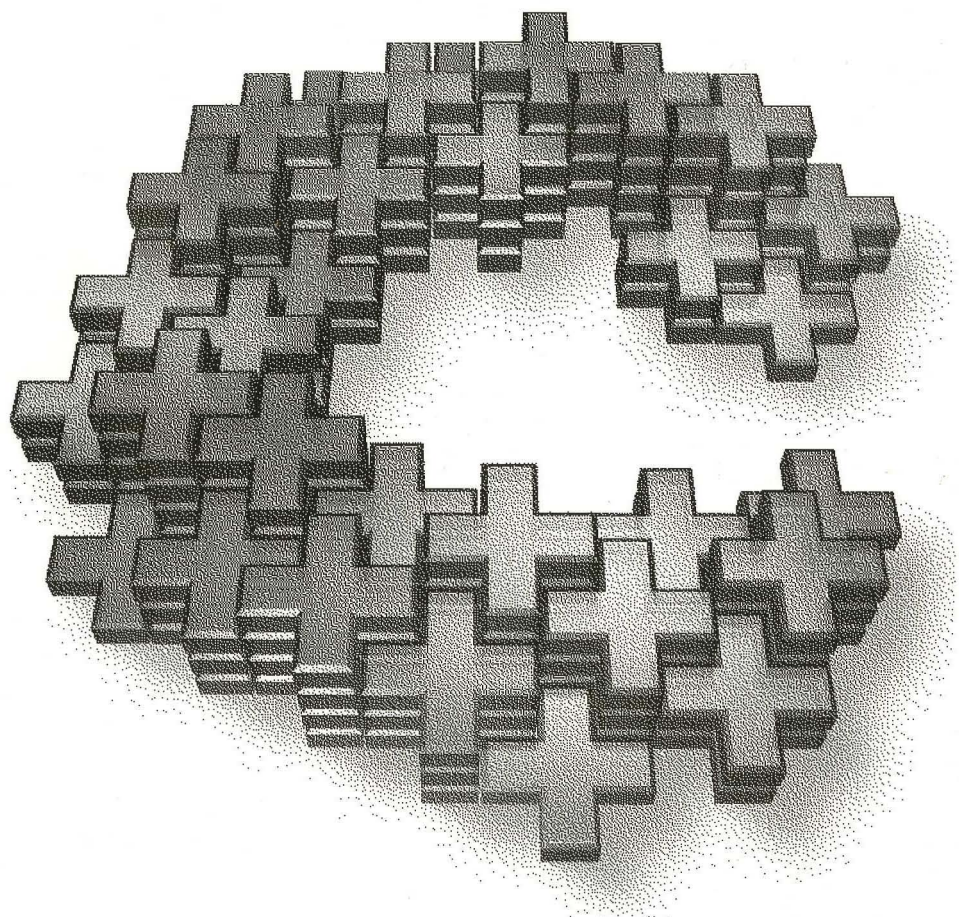
Microsoft®

Visual C++ Tutorials

Development System for Windows 95 and Windows NT



Tutorials

Microsoft® Visual C++™

Version 4.0

Development System for Windows® 95 and Windows NT™

Microsoft Corporation

Document No. DB65885-0995
Printed in the United States of America

Contents

Part 4 The Database Tutorials

Part 6 Appendix

Figures and Tables

Figures

Tables

Introduction

The Microsoft® Visual C++™ 4.0 development system for Windows® and Windows NT™ adds fully integrated Windows-hosted development tools and a "visual" user-interface-driven paradigm to the traditional C/C++ development process.

About This Book

This book is divided into the following parts:

- Part 1, Developing an Application Using Visual C++, provides an overview of the development process and the Visual C++ tools you'll use to build your applications.
- Part 2, the Scribble Tutorial
- Part 3, the OLE tutorials, includes:
  - Container
  - AutoClik
  - Circle
- Part 4, the database tutorials, includes:
  - Enroll
  - DaoEnrol
- Part 5, Win95 Compliance, discusses considerations for meeting the Win95 logo requirements.
- Appendix A provides information about products and services that make the Microsoft Visual C++ development system more accessible for people with disabilities.

The following section, gives an overview of the main topics covered by each Tutorial.

Overview of the Class Library Tutorials

The tutorials included in this book demonstrate practical application of the Microsoft Foundation Class (MFC) Library classes, including the OLE and database classes, both Open Database Connectivity (ODBC) and Data Access Objects (DAO). The class library is a set of C++ classes that encapsulate the functionality of applications written for the Microsoft Windows family of operating systems.

The Scribble Tutorial, introduces the basics of creating MFC applications, and demonstrates how to use the class library's main features. Among the features covered in the Scribble tutorial are the following:

- Using AppWizard to create a skeletal starter application upon which to build your program.
- Implementing a document class to manage your application's data and to write to and read from files.
- Implementing a "view" class to display your document and manage all user interaction with it.
- Using the Visual C++ resource editors to create and edit your application's resources.
- Using WizardBar to connect user-interface objects, such as menu items, buttons, and accelerator keys, to handler functions in your source code.
- Using the Visual C++ dialog editor to create a dialog resource, and ClassWizard to encapsulate the dialog resource in a dialog class. Also, using WizardBar to declare handler functions, map the dialog box controls to member variables of the dialog class, and define validation rules.
- Implementing scrolling and splitter windows; enhancing the default printing capabilities, including Print Preview; and adding context-sensitive Help to your application.

The OLE Server Tutorial is Scribble Step 7, which demonstrates adding OLE in-place editing server support to an existing application.

The OLE Container Tutorial introduces the MFC OLE classes and teaches you to use their main features. Among the features this tutorial describes are the following:

- Creating an OLE in-place editing container starter application by using AppWizard
- Implementing the OLE client item rectangle
- Implementing hit testing and selection
- Implementing activation with the mouse click
- Implementing special cursors for resizing and moving objects

- Drawing embedded objects
- Deleting embedded objects

The OLE Automation Server Tutorial describes how to create and test a simple OLE Automation server application, starting with AppWizard's built-in support for automation. Among the tutorial topics are:

- Analyzing the dispatch interface name
- Analyzing AppWizard-provided code
- Creating an OLE type library
- Implementing properties of a dispatch interface
- Implementing methods of a dispatch interface
- Creating a new **CCmdTarget** class with a dispatch interface
- Referring to one dispatch interface from another

The Circle Control Tutorial introduces the OLE control classes. Among the topics discussed are:

- Building an OLE control
- Implementing background painting behavior
- Adding a custom property
- Implementing Get/Set methods for a custom property
- Adding special effects
- Adding custom events
- Handling text and fonts
- Working with property pages
- Simple data binding
- Versions and serialization of OLE controls

The database tutorials introduce the database classes and describe how to use their main features, demonstrating both ODBC and DAO support. Among the features these tutorials describe are:

- Selecting and registering a database with ODBC, or using the DAO database classes to read the data source directly (if applicable).
- Examining the classes AppWizard generates for your database application.
- Using ClassWizard to bind controls on your form to the member variables of a "recordset" object. A recordset represents a set of records selected from a data source. The framework takes care of moving data between the controls and the data source.

- Using a database form based on two or more recordsets. Each recordset represents a different table or query.
- Controlling the data selected by a recordset. You can filter a recordset, parameterize it at run time, and sort it.
- Adding, editing, and deleting records and implementing commands for these actions.

Adding Windows95 Functionality provides information about making your application Windows 95-compliant, including the Windows 95 support that AppWizard provides for you. Topics include:

- Summary of the logo requirements
- Following UI recommendations
- Using tabbed property pages
- Using common controls
- Displaying a shortcut menu
- Using the system registry
- Creating a Setup and an Uninstall program
- Adding OLE support, including:
  - Being a drop target
  - Providing summary information
- Adding Messaging API (MAPI) support

Document Conventions

This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, registers, and terms used at the operating-system command level.
char, **_setcolor**, **__far**	Bold type indicates C and C++ keywords, operators, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown. Many constants, functions, and keywords begin with either a single or double underscore. These are required as part of the name. For example, the compiler recognizes the **__cplusplus** manifest constant only when the leading double underscore is included.

Example	Description
expression	Words in italics indicate placeholders for information you must supply, such as a filename. Italic type is also used occasionally for emphasis in the text.
[*option*]	Items inside square brackets are optional.
#pragma pack {1 | 2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless square brackets ([ ]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text.
CL [*option...*]*file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.

`while()`
`{`
`.`
`.`
`.`
`}`
	A column or row of three dots tells you that part of an example program has been intentionally omitted.

CTRL+ENTER	Small capital letters are used to indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second.
↵	The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
`"C string"`	Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " and ' ' rather than and ‘ ’.
Dynamic-Link Library (DLL)	The first time an acronym is used, it is usually spelled out.
Microsoft Specific	Some features documented in this book have special usage constraints. A heading identifying the nature of the exception, followed by an arrow, marks the beginning of these exception features.
END Microsoft Specific	**END** followed by the exception heading marks the end of text about a feature that has a usage constraint.
▶ CEnterDlg;	The arrow adjacent to the code indicates that it has been altered from a previous example, usually because you are being instructed to edit it.

Microsoft Support Network

In the event you cannot install Microsoft Visual C++, please refer to the telephone support offerings below. Microsoft's support offerings range from no-cost and low-cost electronic information services (available 24 hours a day, 7 days a week) to annual support plans and CD-ROM subscription programs. Please check the Technical Support section in online Help for detailed information.

Microsoft support services are subject to Microsoft's then-current prices, terms, and conditions, which are subject to change without notice.

Standard Support

No-charge support from Microsoft support engineers is available via a toll call between 6:00 A.M and 6:00 P.M. Pacific time, Monday through Friday, excluding holidays. In Canada, call between 8:00 A.M and 8:00 P.M. Eastern time, Monday through Friday, excluding holidays. This support is available for 30 days after you make your first call.

- In the United States, for technical support for Microsoft Visual C++, call (206) 635-7007.
- In Canada, for technical support for Microsoft Visual C++, call (905) 568-3503.

When you call, you should be at your computer and have the appropriate product documentation at hand. Be prepared to give the following information:

- The version number of the Microsoft product that you are using
- The type of hardware that you are using, including network hardware, if applicable
- The exact wording of any messages that appeared on your screen
- A description of what happened and what you were doing at the time
- A description of how you tried to solve the problem

Priority Support

The Microsoft Support Network offers priority telephone access to Microsoft support engineers 24 hours a day, 7 days a week, excluding holidays, in the U.S. In Canada, the hours are from 6:00 A.M to midnight, 7 days a week, excluding holidays.

- In the United States, call (900) 555-2300; $2.95 (U.S.) per minute, $95 maximum. Charges appear on your telephone bill. Not available in Canada.
- In the United States, call (800) 936-5800, at $95 (U.S.) per incident, billed to your VISA, MasterCard, or American Express card.
- In Canada, call (800) 668-7975 for more information.

Text Telephone

Microsoft text telephone (TT/TDD) services are available for the deaf or hard of hearing. In the United States, using a TT/TDD modem, dial (206) 635-4948. In Canada, using a TT/TDD modem, dial (905) 568-9641.

Product Support Worldwide

The following list contains Microsoft subsidiary offices and the countries they serve. If there is no Microsoft office in your country, please contact the establishment from which you purchased your Microsoft product. This list provides only basic technical support phone and fax numbers; other services such as BBS and sales numbers may be available. For additional subsidiary information, check the Product Support Services Worldwide section in online Help.

When you call, you should be at your computer and have the appropriate product documentation at hand. Please follow the guidelines listed above under "Standard Support."

Area	Telephone Numbers	Fax Numbers	Area	Telephone Numbers	Fax Numbers
Argentina	(54) (1) 815-1521	(54) (1) 814-0372	Liechtenstein	01-342 4036	01-831 08 69
Australia	(61) (02) 870-2131	(61) (02) 805-0519	Luxembourg	(32) 2-5133274	
Austria	0222-68 0660-6515	0222-68 16 2710	Mexico	(52) (5) 325-0912	
Belgium	02-51 33274		Netherlands	02503-77877	
Bolivia	(54) (1) 815-1521	(54) (1) 814-0372	New Zealand	64 (9) 357-5575	64 (9) 358-3726
Brazil	(55) (11) 871-0090	(55) (11) 240-2205	Northern Ireland	(01734) 270007	(01734) 270080
Caribbean	(214) 714-9100	(809) 273-3636	Norway	(47) (22) 02 25 50	(47) (22) 02 25 70
Chile	56 2 232-4467	56 2 233 5917	Papua New Guinea	(61) (02) 870-2131	(61) (02) 805-0519
Colombia	(571) 618 2255	(571) 618 2269	Paraguay	(54) (1) 815- 1521	(54) (1) 814-0372

Area	Telephone Numbers	Fax Numbers	Area	Telephone Numbers	Fax Numbers
Czech Republic	(+42) (2) 2683-20 or -27	(+42) (2) 266020	Poland	(+48) (2) 6216793 or (+48) (71) 441357	(+48) (2) 6615434
Denmark	(45) (44) 89 01 11	(45) (44) 89 01 44	Portugal	(351) 1 4412205	(351) 1 4412101
Dubai	(971) 4 513 888	(971) 4 527 444	Republic of China	(886) (2) 508-9501	(886) (2) 504-3121
Ecuador	(593) (2) 463-094		Republic of Ireland	(01734) 270007	(01734) 270080
England	(01734) 270007	(01734) 270080	Russia	(+7) (095) 267-8844 or 158-6963	(+7) (502) 224 50 45
Finland	(0358) (90) 525 502 500	(46) (0) 8 752 29 00	Scotland	(01734) 270007	(01734) 270080
France	(33) (1) 69-86-10-20	(33) (1) 64-46-06-60	Singapore	(65) 220-7202	(65) 227-6811
French Polynesia	(33) (1) 69-86-10-20	(33) (1) 64-46-06-60	Slovenia	(+386) (61) 1232354	
Germany	089- 3176-1150	089- 3176-1000	Slovak Republic	(+42) (7) 312083	(+42) (2) 266020
Greece	(30) (1) 6893-631 through 6893-635	(30) (1) 6893 636	South Africa	0 802 11 11 04	(27) 11 445 0045 or (27) 11 445 0046
Hong Kong	(852) 804-4222	(852) 560-2217	Spain	(34) (1) 803-9960	(34) (1) 803-8310
Hungary	(36) (0) 1/1172289	(+36) (1) 269 1030	Sweden	(46) (8) 752 09 29	(46) (0) 8 752 29 00
India	(01) (91) 646-0694 (01) (91) 646-0767	(01) (91) 646-0813	Switzerland	01-342 40 36	01-831 08 69
Ireland	(01734) 270007	(01734) 270080	Turkey	(90) 212 2585998	(90) 212 2585954
Israel	972-3- 613-0833	972-3- 613-0834	United Kingdom	(01734) 270007	(01734) 270080
Italy	(39) (2) 7039-8351	(39) (2) 7039-2020	Uruguay	(54) (1) 815-1521	(54) (1) 814-0372
Japan	(81) (3) 5454-2364	(81) (3) 5454-7972	Venezuela	58.2.910046 58.2.910510	58.2. 923835
Korea	(82) (2) 531-4800	(82) (2) 531-1724	Wales	(01734) 270007	(01734) 270080

PART 1

Developing an Application Using Visual C++

CHAPTER 1

Developing an Application Using Visual C++

This section describes how you'll use the Microsoft Developer Studio to develop an application, and points to sources within the Visual C++ documentation set where you may find more information including, but not limited to, the tutorials that comprise the rest of this book.

The tutorials show you how to build sample applications, while demonstrating key concepts you'll need to understand when developing your own Visual C++ applications. If you're new to Visual C++ and Microsoft Developer Studio, the tutorials are a great hands-on way to become more familiar with it.

For conceptual information on using the Microsoft Foundation classes, see *Programming with MFC*. The complete documentation set for Microsoft Visual C++ and Microsoft Developer Studio fully describes the integrated development environment, new features and tools, and ways to optimize your work.

The Development Process

Aside from the undefinable aspects of creating an application — inspiration, hard work, frustration and dedication, to name a few —there are several stages that can be broadly categorized. These stages are outlined here, and briefly discussed within the context of Visual C++ and Microsoft Developer Studio. Each programmer may approach any given stage at a different time or in a different order; no preferred order is implied.

Creating a New Project

Generally, you'll use AppWizard to generate a robust set of application "starter" files. Within AppWizard, you specify the structure you'd like your application to take, and the options and features you'd like AppWizard to provide.

Choosing a Project Type

You can use AppWizard to generate the following types of MFC projects:

- Executable
- DLL
- OLE Control
- Custom AppWizard

In addition, you can create many non-MFC project types, such as console applications, static libraries, and projects based on external makefiles. For more information on building projects based on external makefiles, see "Using External Projects." For more information on building a single file as a project, see "Building a Single File without a Project Workspace."

For more information about working with projects, see "Working with Projects."

Choosing an Application Type

Once you've chosen your project type, you'll choose the application type. AppWizard provides the following types to choose from, with MDI being selected as the default.

- Single Document Interface (SDI)

  The Enroll Tutorial is an SDI-based application.

- Multiple Document Interface (MDI)

  The Scribble Tutorial demonstrates creating a simple MDI application.

- Dialog-based

Specifying Initial Application Features

After choosing a basic project and application type, your choices become more specific. AppWizard provides branching options you can use in various combinations to create the basis for sophisticated applications. For example, based on your specifications, AppWizard will generate MFC code for any of the following features:

- Standard menus and dialog boxes for opening, editing, arranging and saving files, previewing print jobs, and printing.
- ODBC and DAO support.

  The Enroll Tutorial demonstrates building a simple database application with ODBC support. The DaoEnrol Tutorial demonstrates the Enroll application using the DAO classes.

- Support for OLE, including support for:
  - OLE compound documents, both server and container.

    Scribble Step 7 demonstrates adding OLE server support to Scribble. The OLE Container Tutorial demonstrates building an OLE container application.
  - OLE compound files.
  - OLE automation.

    The AutoClik Tutorial demonstrates building an OLE Automation server.
  - OLE controls.

    The Circle Tutorial demonstrates building an OLE control.
- A dockable toolbar, status bar, and 3D controls.
- Windows Open System Architecture (WOSA) support including the Messaging API (MAPI) and Windows sockets.
- Support for context-sensitive Help, including a menu and toolbar button. In addition, AppWizard provides initial .RTF (help project source) files.

  Scribble Step 6 is a complete tour of providing context-sensitive Help for your Visual C++ applications.

Custom AppWizards

If these options aren't as tailored to your needs as you'd like, you can design a Custom AppWizard that, just like AppWizard, generates a set of starter project files. With Custom AppWizard, however, you can specify the basis for the application according to your needs, your company's particular data sources, and so forth. Then, every time you run this custom AppWizard, it generates the starter files according to those specifications.

For more information about creating a custom AppWizard, see "Creating Custom AppWizards."

Component-Based Development

The Visual C++ Component Gallery makes building powerful applications more straight-forward than ever. By simply selecting from the many pre-built components and inserting them into your project, you can add fully-developed features to your application, such as owner-draw controls, pop-up menus, OLE controls and more. The Component Gallery also significantly increases code reuse in your application development process.

For more information about Component Gallery, see "Using Component Gallery."

Adding and Editing User-Interface Objects

A large part of developing and refining your interface involves customizing user-interface objects, or Windows resources. Visual C++ provides a rich set of resources you'll use to enhance the initial interface that AppWizard creates for you. By adding various resources to your project, and editing them in the intuitive Visual C++ resource editors, you can quickly implement your user-interface design.

The Visual C++ resource editors enable you to move easily between design mode and a run-time view of interface objects, such as menus, toolbars and dialog boxes. For more information, see "Using the Resource Editors."

Connecting User-Interface Objects with Code

The MFC code that AppWizard generates may be sufficient for a small number of your application's user-interface objects, such as the splash screen, but for the most part you'll have application-specific routines that you want called when an interface object receives input from the user. To do that, you need to make a connection between the interface object, the event, and the code you want called.

The visual tools provided by Visual C++ 4.0 simplify the task of connecting user-interface objects to your application-specific code. If you've used Visual C++ before, you're probably familiar with the ways that ClassWizard automates such tasks as generating message-handler functions and message maps. Version 4.0 implements two short-cuts to the ClassWizard dialog: ClassView and WizardBar. In addition to providing a seamless link between your project classes and files, these tools make navigating through your source code as easy as pointing and clicking.

- ClassView

  ClassView displays your classes and their members in a hierarchical view, using visually intuitive icons. For instance, it displays private member functions with a "padlock" next to the function name. By simply clicking a class or any of its displayed members, you can jump directly to the associated declaration or definition in your project source file.

  ClassView provides a pop-up menu that you can use to directly add member functions and variables to a given class. (You'll still use ClassWizard, or WizardBar, to add message-handling functions.)

  You can also use ClassView to:

  - Browse symbols
  - Display graphs of symbol relationships
  - Set breakpoints
- WizardBar

Like ClassView, WizardBar enables you to jump directly to a member function. Additionally, you can use WizardBar to override virtual functions, and to create a handler for the Windows messages or commands a particular class can respond to.

The tutorials demonstrate how these features simplify and speed the process of specifying how your user-interface objects respond to messages.

Editing Code

You'll continue to use the navigational support provided by ClassView and WizardBar as you edit your project source code. The Visual C++ integrated text editor provides full-featured editor support. In addition, version 4.0 offers the following new editor features:

- Emulation modes for either BRIEF or Epsilon
- Incremental searching

  For more information, see "Finding and Replacing Text."
- Automatic indentation and separate tabs and indent sizes

  For more information, see "Setting Tabs and Indents."
- Full-screen editing mode
- New navigating commands

For more information about editing your source code, see "Using the Text Editor."

Compiling and Building Your Project

You'll compile and build at every step of project development, to check that your code is correct and that the your application behaves as you expect it to. In Visual C++ 4.0, the initial default project configuration builds your application with debugging information, so that you can easily debug any errors you discover. You can specify a different project configuration to be built by default. For more information, see "Setting the Default Project Configuration."

Visual C++ 4.0 provides several enhancements to the build process, including:

- Building subprojects, such as DLLs
- Specifying custom build settings for alternate file types, such as Help project (.HPJ) files

For more information, see "Building a Project Configuration."

In addition, Visual C++ 4.0 offers many ways to optimize the build process, including:

- Minimal rebuild
- Incremental compilation
- Improved incremental linking

For more information, see "Setting Compiler Options and Setting Linker Options," in the *Visual C++ User's Guide*.

Debugging the Application

Compiling and building leads naturally to debugging. You may want to debug your application when a build failure occurs, or when you've modified your source code. You can run a program in the Visual C++ debugger, if you've built the debug project configuration. Or, you can enter Debug mode if your application fails in execution, without closing the application.

The Visual C++ 4.0 integrated debugger adds features that make debugging a less troublesome process including:

- DataTips
- Variables window
- Stepping into a specific function

  For more information, see "Stepping Into Functions."
- QuickWatch
- New Watch window
- TCP/IP remote debugging

For more information about using the Visual C++ Debugger, see "Using the Debugger."

PART 2

The Scribble Tutorial

CHAPTER 2

Scribble Tutorial

It's a traditional programming practice to begin work with a new system or language compiler by writing a program that prints "Hello, World!" on the display. When you begin programming in a graphical user-interface (GUI) environment such as Microsoft Windows, however, the traditional practice is hard to follow. There's a fair amount of programming overhead—well in excess of the few lines of "Hello, World!"—simply to get a minimal GUI application running.

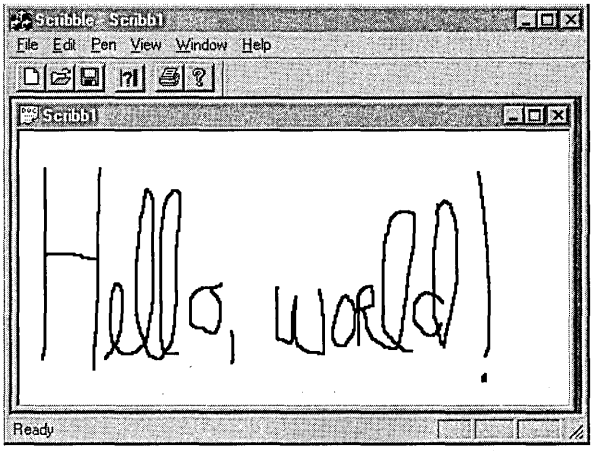
Note This tutorial is designed for Win32. If you have Visual C++ for Macintosh, you will see some Macintosh-specific resource IDs that have "$(_MAC)" appended to the name. For the purposes of this tutorial, you can either ignore or modify them in parallel with the Win32 resource IDs of the same name.

Scribble, the application you build in this tutorial, is a tiny drawing program that poses a more realistic trial run in the Windows programming environment than "Hello, World!" Instead of printing that little phrase so familiar to programmers, Scribble lets the user *draw* "Hello, World!" (or any free-hand drawing) by using the mouse, and then save the image in a file.

By the end of the tutorial, Scribble has custom menus, MAPI support, a dialog box with automatic initialization and validation, printing and print preview, scrolling, splitter windows, application Help support, and more. That's a fitting list of features for a GUI "Hello, World!" As you'll see, Microsoft Developer Studio provides many of these features "for free," and makes implementing the rest easy.

Figure 2.1 shows what Scribble looks like on the screen.

Figure 2.1 Scribble in Action



Installing the Sample Files

If you want to examine source code for Scribble, or any other tutorial sample you choose to create by following the steps in this book, you can install the sample files that are included on the Microsoft Developer Studio CD. You don't need to install the sample files in order to build the tutorials yourself; simply follow the directions in the tutorial chapters. You might find it useful, however, to have the sample source files available locally, for example if you want to compare code between the sample files and your version of the tutorial project.

There are also many samples included on the CD that are not covered in the tutorial chapters. You can easily install the files needed to build these sample programs as well.

▶ To install sample files

1 From InfoView, expand the following folders:

Samples \ MFC Samples \ Tutorial Samples

Under the Tutorials book icon, you'll see page icons that represent each of the tutorial chapters in this book.

(Non tutorial samples are also located under the Samples \ MFC Samples category, organized according to topic. For example, Samples \ MFC Samples \ MFC OLE Samples contains several sample applications that demonstrate different aspects of OLE support.)

2 Choose the page icon for the application you are interested in, for example, Scribble.

In the topic window, you'll see an overview topic for the tutorial or other sample application you chose.

Use the buttons provided toward the end of the overview topic to install the sample files. For the tutorial topics, each tutorial step is represented by a separate button.

3 Click the button for the tutorial step or other sample application you want installed locally.

4 In the Sample Application dialog box, choose the Copy All button to install all files needed to build and run the sample application or tutorial step.

You can also select and copy one or more individual files if you wish; or use the View button to examine a file before copying it. When you choose either the Copy or Copy All button, a dialog appears in which you can specify the directory where you want the sample files installed.

5 Specify the directory for the sample files, and choose OK.

When the sample files have been copied, you are returned to the Sample Application dialog box.

6 Choose Close.

You can also use Books Online to preview sample applications.

A Note About Long Filenames

The names of the source code files on the distribution CD (and therefore in Books Online) comply with ISO standards, which specify an 8.3 filename configuration. When you follow the instructions in this book to generate your tutorial project files, AppWizard generates long filenames by default.

If you use the project files from the distribution CD or from Books Online (the sample source files) as your starting point to develop a particular tutorial step, you may notice discrepancies between the filenames. Class names remain the same. When the tutorial instructions mention a project filename, for example, ScribbleView.cpp, use the corresponding short filename, for example ScribVw.cpp, when performing any operations in the file.

Scribble Step 7

For Scribble Step 7 (Creating an OLE Server), it's probably best not to start from the sample source files. So long as you use your own Step 6 files as the starting point for Scribble Step 7, the project filenames will remain compatible.

If you do choose to use the sample source files, when the tutorial steps instruct you to change a filename, you must substitute the short filename in order to avoid naming conflicts. For instance, **#include** statements may refer to the short filename rather than the long filename AppWizard generates. For more information, refer to "Using Short Filenames" in Chapter 12.

Previewing the Sample Applications

Without needing to install any sample files, you can use Books Online to preview sample applications, including tutorial samples, in their executable state. For example, if you're simply reading through the Scribble tutorial without adding code, you can still run Scribble at each step to see what it looks like and how it behaves.

▶ To run a sample application from the source files

1 From InfoView, expand the following folders:

Samples \ MFC Samples \ Tutorial Samples

Under the Tutorials book icon, you'll see page icons that represent each of the tutorial chapters in this book.

(Non tutorial samples are also located under the Samples \ MFC Samples category, organized according to topic.)

2 Choose the page icon for the application you are interested in, for example, Scribble.

This brings you to a topic window that corresponds to the sample file (or tutorial step) that you want to preview.

3 From the topic window, choose the appropriate button. For example, to preview Scribble as it appears after compiling Step 2, choose the Step 2 button.

4 In the Sample Application dialog box, select the executable file.

Notice that the View button in the dialog box changes to a Run button.

5 With the executable file selected, choose Run.

Developer Studio launches a version of the application that you can use to view and test the application's functionality. When you are finished previewing the application, close it as you normally would.

You are returned to the Sample Application dialog box.

6 Choose Close to exit the dialog box.

Scribble Tutorial Steps

The tutorial develops the Scribble application in seven steps. By installing the sample files, you can make a local copy of all project files in their representative state for each particular step. By default, these files are copied to subdirectories (represented as folder icons in your Project Workspace window) named STEP1 through STEP7. You can change the name and location of the subdirectories if you like.

Chapter 3 describes how to create the skeleton AppWizard files for Scribble. Chapters 4 through 11 describe how to develop the basic Scribble application (Steps 1 through 6). Chapter 12 describes how to convert Scribble to an OLE server (Step 7).

Note that a single tutorial step might be covered in more than one chapter. For convenience, Table 2.1 correlates chapters, steps, and chapter content. The second column gives the step completed by the end of the corresponding chapter. Each chapter begins where you left off in the previous step.

Table 2.1 Tutorial Steps

Chapter	Step Completed	Content
3		Starter AppWizard application (no corresponding tutorial step); includes MAPI support
4		Scribble's document (first part of Step 1)
5	1	Scribble's view (completes Step 1)
6		Menus and toolbar–menu and graphics editors (first part of Step 2)
7	2	Handlers for commands–ClassWizard, WizardBar, Classes pane (completes Step 2)
8	3	Dialog boxes–resource editors, ClassWizard, WizardBar, Classes pane, adding a spin control
9	4	Scrolling and splitting
10	5	Printing and print preview
11	6	Context-sensitive Help
12	7	OLE server creation

The Files You Work With

For most of the procedures that follow, you will need to deal with only a few of the files actually generated by AppWizard:

- Document class files: ScribbleDoc.h (the header file) and ScribbleDoc.cpp (the implementation file)
- View class files: ScribbleView.h and ScribbleView.cpp

You'll also occasionally refer to or edit Scribble.h and Scribble.cpp, the application class files.

Tip ClassView makes working with your class files easy and intuitive. By expanding the top-level folders, you can display all member functions contained in the class. Then, by choosing the icon for the member function you wish to edit, you can go directly to the file that contains the function. You needn't explicitly open the file, and you can begin editing immediately, because Developer Studio places your cursor at the selected function. For more information, see "Navigating Through Code."

For chapters that use the resource editors (Chapters 5 through 7), you'll work with Scribble.rc, the application's resource file. This file and the resources it contains (such as Windows bitmaps and dialog resources) are represented in ResourceView. For more information, see "Viewing Resources," in Chapter 4 of the the *Visual C++ User's Guide*.

You may occasionally want to examine the other files created by AppWizard and ClassWizard, but in most cases you won't need to alter them. For more information about the skeleton application files that AppWizard creates, see the ReadMe.txt file in your Scribble project root directory (AppWizard also creates this file). More detailed information is available in the article "AppWizard: Files Created," in Part 2 of *Programming with MFC*.

In Chapter 12, "Creating an OLE Server," you'll work with the files for MFC classes that implement OLE server support.

Note For Chapter 3, "Creating a New Application" with AppWizard, simply follow the instructions for using AppWizard to create the skeleton Scribble application files. Even if you aren't planning to add the tutorial code yourself, you can easily follow this procedure. It's a good way to learn to use this tool.

Scribble Build Information

This section explains a few things you'll need to know when you prepare to build Scribble. General procedures for compiling and linking MFC programs in Visual C++ and running the executable program under Windows are given in Chapter 1, "Developing a Microsoft Visual C++ Application."

If you're working along, adding code as you read, build the version of the project that you've been developing. You should be in your Scribble project directory and have the project open. Setting up the project directory is described in Chapter 3, in the procedure, "To create starter files for Scribble."

Setting Build Options

When you create a new project, Developer Studio automatically creates both Release and Debug build configurations, although only one project configuration (the current default) is displayed in the Project Workspace window. Developer Studio sets Debug as the default project configuration. For Scribble, you shouldn't need to change this default setting. However, for your own projects you'll want to be able to build a Release version.

▶ To select Debug or Release build options

- From the Default Configuration drop-down list on the Build toolbar (just above the Project Workspace window), select the type of project you want to build.

  –or–

- From the Build menu, choose Set Default Configuration, and in the dialog box, select the type of project you want built by default.

  The project you select remains the default project until you change it again.

Chapter 3, "Creating a New Application with AppWizard," begins the tutorial proper. You'll use AppWizard to create the skeleton Scribble application. Then, in the chapters that follow, you'll build a more powerful Scribble application upon that skeleton.

CHAPTER 3

Creating a New Application with AppWizard

Once you've completed your initial application design, you'll typically perform the following tasks to develop the application with Microsoft Developer Studio and the Microsoft Foundation Class Library (MFC):

- Use AppWizard to create a set of C++ starter files and associated Windows resources — a skeleton application that you can build and run immediately.

  For more information, see Chapter 1, "Creating a New Application Using AppWizard," in the *Visual C++ User's Guide.*
- Use resource editors to construct the objects that make up the user interface, such as menus and dialogs.
- Use elements of the IDE to generate and edit application-specific code. These elements include, but are not limited to:
  - The text editor
  - ClassView
  - ClassWizard
  - WizardBar
- Build, browse, test, and debug your project files — then add more code.

The steps tend to be iterative. You'll probably go back and forth between editing the user interface and writing code all through the development process. You can also do the steps in a different order, depending on your working style.

Starting with Scribble

This chapter shows you how to create the starter files for the Scribble application that is developed throughout the tutorial. As mentioned, these files contain skeletal code for several C++ classes:

- An application class
- A document class

- A view class
- A frame window class

The concepts behind these classes are discussed fully in Chapter 1, "Using the Classes to Write Applications for Windows," of *Programming with MFC*.

You'll also learn more about them in Chapters 8 and 9 of this manual.

Details about the files AppWizard creates are available in a text file (ReadMe.txt) that is created along with the starter files. For additional information about the starter files, see the article "AppWizard: Files Created" in Part 2 of *Programming with MFC*.

Without adding a line of code, you can build the starter application you created with AppWizard and run the resulting program, which exhibits much of the standard functionality you expect from a program written for Windows. The steps needed to build and run the program are given in the sections "Build the Starter Application" and "Run the Starter Application."

If you're working along, adding code as you read, follow all directions in this chapter. When you finish, you'll have a full set of starter files in your own project directory. If you're reading along without adding any code, it's still a good idea to work through this chapter to familiarize yourself with AppWizard.

Note If you do not have the Sample files in your current Microsoft Developer Studio installation, you can easily install them. For more information, see "Installing the Sample Files." Note that the filenames in the sample source code may differ from the ones generated by AppWizard, depending on your specifications.

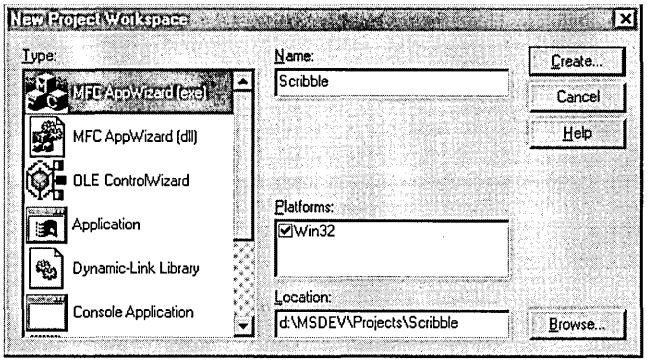
Using AppWizard to Create the Starter Application for Scribble

This section shows you how to use AppWizard to create the starter application that forms the beginnings of Scribble.

AppWizard presents a series of dialog boxes that contain options you specify, depending on your preferences. Then AppWizard creates a set of source-code files, based on these options, from which you develop your application. This saves a great deal of time and effort and lets you focus on the application-specific parts of your program.

Note The following procedure describes how to enter the correct values for Scribble. Many of the AppWizard dialog boxes contain choices that you won't use to create the starter files for Scribble. For more information on these choices, see Chapter 1, "Creating a New Application Using AppWizard," in the *Visual C++ User's Guide*.

Figure 3.1 New Project Workspace Dialog Box



▶ To create starter files for Scribble

1 From the File menu, choose New.

The New dialog box appears.

2 Select Project Workspace and click OK.

The New Project Workspace dialog box (Figure 3.1) appears.

3 In the Name box, type `Scribble`.

AppWizard creates a project directory with this name under the main (root) directory specified in the Location box. The workspace configuration file and project makefile are based on this name, in this case, Scribble.mdp and Scribble.mak, respectively.

4 In the Type listbox, make sure MFC AppWizard (exe) is selected.

5 If necessary, use the Location box to specify a different root directory for the Scribble project files that AppWizard creates under the Scribble project directory.

Depending on the directory in which you last worked, you may want to change where the Location box currently points to. You can use the Browse button to navigate to an existing directory, or type a directory name directly into the Location box. AppWizard creates this directory if it doesn't exist.

6 If any checkboxes other than Win32 appear in the Platforms box, clear them.

If your Microsoft Developer Studio installation includes other language packages, they are represented by checkboxes as potential platforms. Choosing checkboxes other than Win32 might mean that options you need for Scribble are not available. For more information, see the topic, "Platform Types," in Chapter 2 of the *Visual C++ User's Guide*.

7 Choose the Create button.

AppWizard creates the project directory, and the MFC AppWizard–Step 1 dialog box appears.

8 Choose the Next button in the dialog boxes for AppWizard Steps 1, 2, and 3 to accept the default options.

By default, AppWizard generates source code that supports the multiple document interface (MDI), creating Scribble as an MDI application.

9 In the AppWizard–Step 4 dialog box, check the MAPI (Messaging API) support checkbox.

Note MAPI support is available to users only if they have an electronic mail system.

With the default options in the Step 4 dialog selected, AppWizard creates code that supports the following features:

- A dockable toolbar
- A status bar
- Printing and print preview
- 3D controls

10 Choose the Advanced button.

The Advanced Options dialog box appears with the Document Template Strings page tab selected.

11 In the File Extension box, type `scb`.

This step is optional. AppWizard appends the extension you specify here to the names of files that the user saves with Scribble. If you choose not to specify an extension, users must specify their own file extensions when saving files.

12 In the Filter Name box, change the filter description to read "Scribble Files (*.scb)".

This step is optional. In the compiled Scribble application, this string, for example, "Scribble Files (*.scb)", appears in the List Files of Type: box in the File Open and File Save As dialog boxes.

13 Choose Close.

14 Click the Finish button in the dialog box for AppWizard–Step 4.

The default options in the Step 5 dialog box supply comments throughout the AppWizard-created files, indicating where you need to add your own code.

The default settings in the Step 6 dialog box specify program file and class names. For the purposes of Scribble, you won't modify class or filenames that AppWizard automatically generates.

Note When you follow these steps, the names of the source code files for Scribble differ slightly from the names generated by AppWizard. This may be helpful, when examining source code, in distinguishing between your files and the samples.

The New Project Information dialog box appears, summarizing the settings and features AppWizard will generate for you when it creates your project. You might

want to take a moment to examine the application type, classes, and features that AppWizard automatically provides.

15 Click the OK button in the New Project Information dialog box.

AppWizard creates all necessary files and opens the project.

Viewing the Project

Each pane of the Project Workspace (other than InfoView) displays a different view of the project. You can switch among them to see what AppWizard generated for the starter Scribble project.

- ClassView displays a graphical representation of the Scribble classes and members that AppWizard created.
- FileView displays a list of the Scribble project files that AppWizard generated.
- ResourceView displays the resources AppWizard generated for Scribble.

Project Build Information

AppWizard creates both Debug and Release versions of each project it creates, selecting the Debug version as the default. The current default project configuration is displayed in the Default Configuration drop-list on the Build toolbar. You can switch the default project to specify which configuration you want to build. For more information, see "Build the Starter Application."

Viewing the Starter Classes

Switch to ClassView, and examine the classes that AppWizard defined in the skeleton application files:

- `CAboutDlg`—the class used to create the "About Scribble" dialog box
- `CChildFrame`—the class used to create the MDI child frame window (which contains the Scribble documents)
- `CMainFrame`—the class used to create the MDI main frame window (which contains the child frame windows)
- `CScribbleApp`—the main application object
- `CScribbleDoc`—the document class
- `CScribbleView`—the view class

By expanding each class icon, you can view the default member functions and variables AppWizard declared for you. In most cases, these are skeleton functions that you'll fill in later.

Developer Studio provides several tools that make jumping directly to your code, and generating skeleton handler functions, easy and intuitive. For instance, you can easily jump from ClassView directly to class and member function declarations and

definitions. For more information, see the next section, "Navigating Through Code," or "Using ClassView" in Chapter 2 of the *Visual C++ User's Guide.*

You'll undoubtedly want to examine the source code files. To orient you, AppWizard creates a text file, ReadMe.txt, in your new project root directory. This file explains the contents and uses of the other new files created by AppWizard. Further details are available in the article "AppWizard: Files Created" in Part 2 of *Programming with MFC.*

The final two sections of this chapter guide you through the process of compiling the starter application and running the resulting program to examine its capabilities.

Navigating Through Code

You can use ClassView to quickly jump to existing code. Without having to think about which file to open, you can jump to existing class definitions, member function declarations and definitions, and member variable definitions.

You can use WizardBar to create a new function handler or to jump to existing handler code.

In the event that you want to go to a section in your file that you can't jump to directly, you can use FileView to easily open the file for editing without going through a menu.

▶ To use ClassView to locate class definitions

- From ClassView, double-click the icon for the class whose definition you want to view or modify.

  –or–

1. Point your cursor at the class icon, and click the right mouse button. (The class name must be selected when you click.)
2. In the pop-up menu, choose Go To Definition.

   The associated header (.h) file for the class opens in the text editor with the cursor placed at the beginning of the class definition.

▶ To use ClassView to locate member function declarations

1. In ClassView, expand the folder for the class that contains the member function declaration you want to view or edit.
2. Point your cursor at the icon (shaped like a document page) for the member function, and click the right mouse button. (The member function name must be selected when you right-click.)
3. In the pop-up menu, choose Go To Declaration.

   The associated header (.h) file opens in the text editor with the cursor at the declaration for the function you chose.

▶ To use ClassView to locate member function definitions

1 In ClassView, expand the folder for the class that contains the member function definition you want to view or edit.

2 Double-click the icon that represents the member function definition. Icons for functions are shaped like a document page.

The associated implementation (.cpp) file opens in the text editor with the cursor at the definition for the function you chose.

▶ To use ClassView to locate member variable definitions

1 In ClassView, expand the folder for the class that contains the member variable you want to view or edit.

2 Double-click the icon that represents the member variable. Icons for variables are shaped like a cube.

The associated project file opens in the text editor with your the placed at the appropriate line in your class definition.

Using WizardBar

The WizardBar appears at the top of the text editor whenever you have an implementation (.cpp) file open in an editor window. When the class associated with the implementation file is selected in the Object IDs list, the Messages drop-down list displays select MFC virtual functions, Windows messages, and **CCmdTarget** procedures for the selected class.

The function names listed in bold represent functions that already have handlers. When you choose any of these functions from the list, ClassWizard jumps you directly to the associated definition in your source files.

When you choose a function without a handler, ClassWizard displays a message box that asks whether you want to create a handler for the function. If you respond Yes, ClassWizard creates a skeleton function with the correct parameters and syntax, highlighting the // TODO comment so you can replace it with your code.

For more information, see "Using WizardBar" in Chapter 14 of the *Visual C++ User's Guide*.

Opening a File for Editing

Sometimes you will need to locate a place in one of Scribble's files that you can't automatically jump to by using ClassView or WizardBar. The FileView pane of the Project Workspace window provides a way to easily open your project files without using a menu command.

FileView displays the main project folder. By expanding this folder, you can view your project files. You can double-click any of these files to open them inside the text editor.

▶ To open a file for editing

1 In FileView, expand the top-level project folder, if necessary, to display the project files.

Implementation files, such as. cpp and .rc files, can be found under the main project folder (in this case, Scribble.exe). Other files, such as header (.h) files can be found under the Dependencies folder.

2 Double-click the file in which you want to work.

The file opens in an editor window.

Build the Starter Application

The starter application you created provides the skeleton of a working application for the Windows operating system. You can choose to build either a Release or a Debug version, because Microsoft Developer Studio generates project files and settings for both versions.

Before compiling, select the type of project you want to build.

▶ To select Debug or Release build options,

- From the Default Configuration drop-down list on the Build toolbar, select the type of project you want to build.

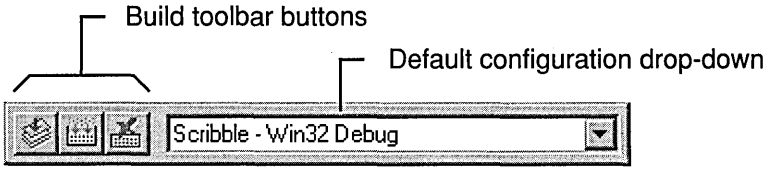
  –or–

- From the Build menu, choose Set Default Configuration, and in the Default Project Configuration dialog box, choose the type of project you want to build.

  The project type you specify in this dialog will remain the default until you change it again.

  The Build toolbar is located just above the Project Workspace window.

Figure 3.2 The Build Toolbar



▶ To build the starter application

- From the Build menu, choose Build Scribble.exe.

  Developer Studio builds the default version of the starter application, producing the file Scribble.exe in your project directory.

When you build the skeleton Scribble application from the files that AppWizard creates— without adding a single line of code — the result is an application that runs, opens, and closes windows, and lets you perform other operations on the windows. Try running the application to view this built-in functionality.

For more information, see "Building a Project," in Chapter 2 of the *Visual C++ User's Guide.*

Run the Starter Application

After you build the starter application, you can run either a Debug or a Release version, so long as you have first built the version of the project you want to run. For more information, see the previous section, "Build the Starter Application," and "Running a Program," in Chapter 2 of the *Visual C++ User's Guide.*

▶ To run the Debug version of Scribble

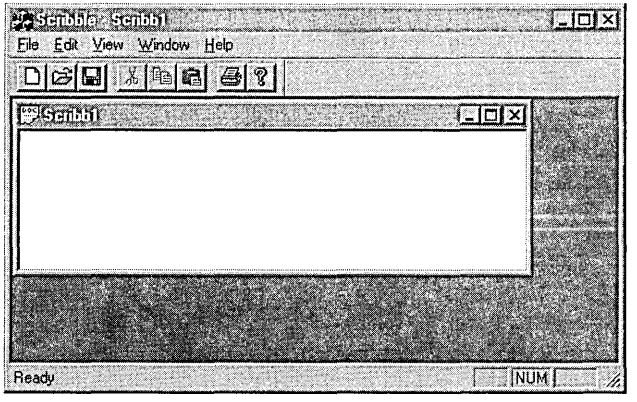
1. From the Default Configuration drop-down on the Build toolbar, select Scribble–Win32 Debug.
2. From the Build menu, choose Execute Scribble.exe.

▶ To run the Release version of Scribble

1. From the Default Configuration drop-down on the Build toolbar, select Scribble–Win32 Release.
2. From the Build menu, choose Execute Scribble.exe.

When the starter application runs, an MDI application window appears with a default toolbar and a menu bar that contains File, Edit, View, Window, and Help menus. The application window contains one open document window, as shown in Figure 3.3.

Figure 3.3 The Starter Application



The document window is empty, of course, because you've added no application-specific code yet. But there is already a great deal of functionality built into Scribble:

- You can use the New command on the File menu to open new windows, and the commands on the Window menu to arrange any open windows.
- The Open, Save, and Save As commands on the File menu are partially functional: at this point, they save empty files. You haven't added all of the code yet to support these commands. The Print command opens the Print dialog box, and the Print Preview and Print Setup commands work.
- MAPI support is fully implemented:
  - The Send command on the File menu works without any additional code from you. (This command, and MAPI support in general, is only available if the user has an electronic mail system.)
- The Recent File command appears on the File menu as a placeholder for the Most Recently Used (MRU) file list, which is implemented automatically when users save and reopen files.
- The commands on the View menu show or hide the toolbar and the status bar.
- The About command on the Help menu brings up an About dialog box with default text in it.
- The default toolbar is partially functional too: the New, Open, Save/Save As, and Print buttons do the same things as the corresponding menu commands, and tool tips appear when your mouse cursor rests over a toolbar button. The status bar at the bottom of the application window displays a description string when you move the mouse pointer over any menu command.

That's a lot of functionality for free! Of course, at this stage the windows have nothing in them. So far, Scribble doesn't scribble.

This application lays the foundation for Scribble and displays much of the standard behavior you expect in an MDI application written for Windows. The next two chapters use Scribble to show you how to develop the document and view classes that you created in this chapter.

For more information about using AppWizard, see Chapter 1, "Creating Applications Using AppWizard," in the *Visual C++ User's Guide*.

CHAPTER 4

Creating the Document

In this chapter and the next, you'll add code to the starter application you created with AppWizard in Chapter 3, "Creating a New Application with AppWizard." You'll be working in the following files: StdAfx.h, ScribbleDoc.h, and ScribbleDoc.cpp.

Note If you have not made a local copy of the sample source code for the tutorial step you're working in, it's easy to do so. For more information, see "Installing the Sample Files."

Developing the CScribbleDoc Document Class

This chapter introduces documents and develops Scribble's document class, called `CScribbleDoc`, an application-specific class derived from class **CDocument**. Chapter 5 introduces views and develops the view class. The two chapters together introduce many of the fundamental concepts of the framework: documents, views, drawing, messages, and serialization (file loading and saving). Because documents and views are intimately related, you need to implement both before Scribble is fully functional.

In later chapters, you'll add new features to Scribble incrementally: more menus, a working toolbar, a dialog box with automatic initialization and validation of its controls, scrolling, splitter windows, enhanced printing, full application Help support, and OLE server support.

Your tour of Scribble's code begins with the starter files created by AppWizard (see the topic "Using AppWizard to Create the Starter Application for Scribble.") You'll add a lot of functionality to Scribble with a small amount of code. Among the things you'll develop in this chapter are:

- Scribble's data—`CStroke`, a class that defines one "stroke" of a drawing.
- Scribble's document—`CScribbleDoc`, a class to contain and manage a list of strokes.
- Scribble's serialization code—code that implements writing and reading documents.

Working Through Scribble Step 1

This chapter and Chapter 5 cover Step 1 of Scribble. If you are working along, begin with the files you created with AppWizard in Chapter 3. As you read this chapter, add or change all lines of code as instructed in the procedures. At the end of Chapter 5, your files should essentially resemble Scribble Step1 source files.

Previewing Step 1

You can easily preview sample applications, including tutorial steps, in their executable state. For more information, see "Previewing the Sample Applications."

Documents

At the heart of every document-based application, such as Scribble, are its document and its view. This section explains the role of the document and introduces Scribble's document class and its members.

At run time, an application written with MFC is a group of cooperating objects that communicate by sending and receiving Windows messages and by calling each other's member functions. Documents are created by document template objects and managed by an application object. Users interact with a document through a view object, which is framed by a document frame window object. Figure 4.1 shows graphically the relationships between these key objects.

Figure 4.1 Objects in Scribble

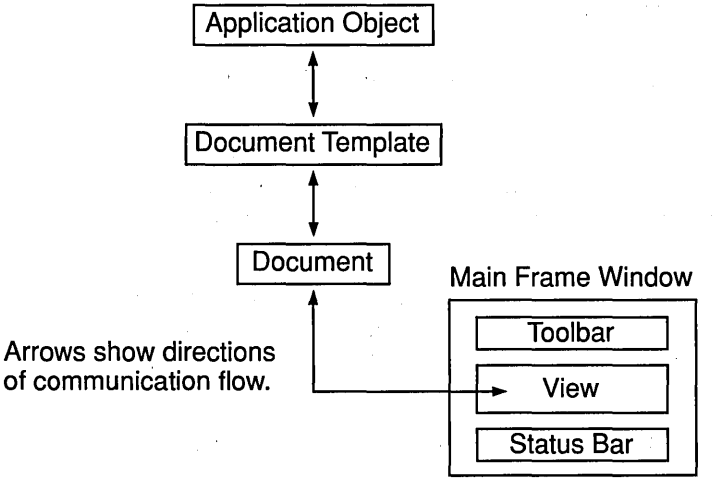


Table 4.1 shows how the document and other objects are created and managed in a framework application.

Table 4.1 Key Objects in an Application

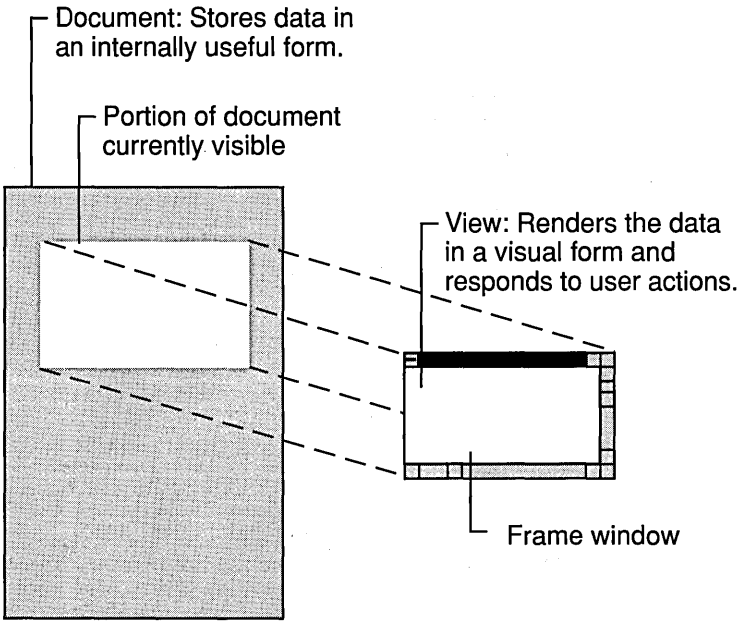
Object	Primary purpose	Relationships to other objects
Application	Manages all other framework objects.	Keeps a list of document templates.
Document template	Creates and manages documents.	Manages a list of open documents of a given type. Creates frame windows and views to provide a user interface for a document's data.
Document	Stores data.	Manages a list of views on its data.
View	Manages user interaction with a document.	Attached to a document. Owned by a frame window.
Frame window	Frames a view.	Owns a view that is attached to a document.

Document Definition

A document is the unit of data that a user opens with the Open command on the File menu and saves with the Save command. The document is responsible for storing the data and for loading it from and storing it to persistent storage, usually a disk file. A document typically appears to the user inside a frame window through which the user manipulates the data.

Figure 4.2 shows the general relationship between a document and its view and frame window.

Figure 4.2 Document and View



Documents in the Framework

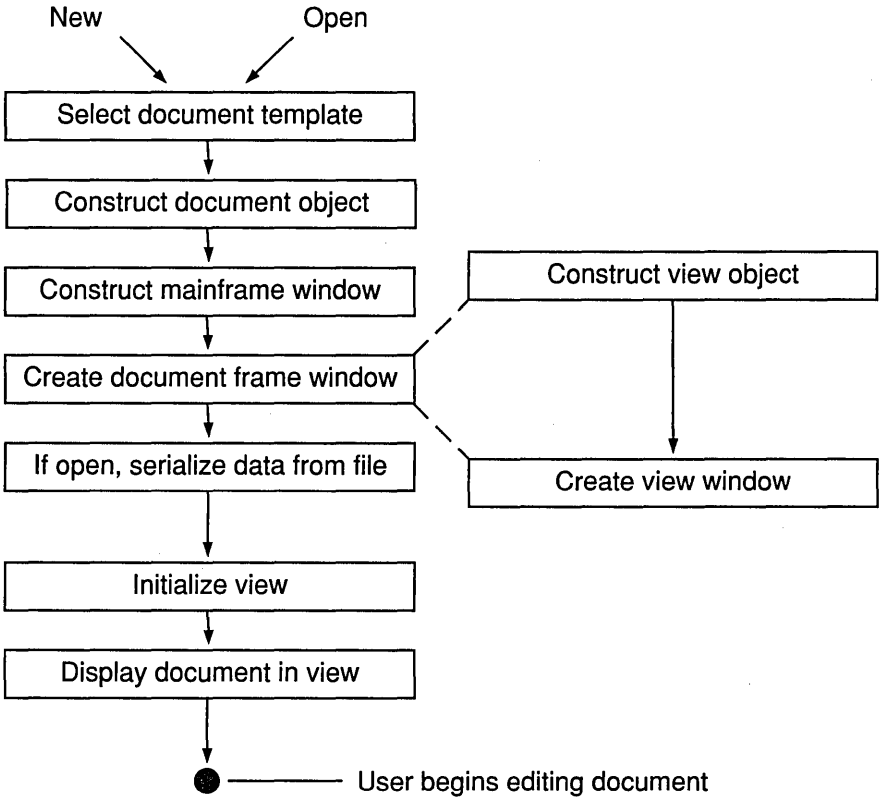
In the framework, the data and the user's operations on the data are managed by two separate objects. A document is an object that stores your data in its member variables, and reads and writes it through a member function called **Serialize**. The user interacts with the document's data through a separate object called a view. The view fills the client area of a frame window, where it displays the data and accepts user input and editing operations. Documents know how to manage data; views know how to display it and accept operations on it. Figure 4.2 shows this important relationship graphically.

Document Creation

When the user opens a document—existing or new—the framework creates a document object and its associated frame window and view objects. If the document is associated with a file, the document reads the file into data structures inside the document. The view obtains data from the document and displays it. Figure 4.3 shows the general process of creating a new document and its view and frame window.

In MFC, documents are based on class **CDocument**. To use **CDocument**, you derive your own document class from it. For more detailed information about documents, see Chapter 1, "Using the Classes to Write Applications for Windows," and Chapter 3, "Working with Frame Windows, Documents, and Views," in *Programming with MFC*, as well as class **CDocument** in the *Class Library Reference*.

Figure 4.3 Creating a Document



How the Document and View Interact

When the user modifies data through the view, the view notifies the document. In turn, the document tells all of its views (a document can have multiple views) to update their displays with the new information, and the views respond by redrawing all or part of the visible portion of the document. You'll learn more about the view's part in this process in Chapter 5, "Creating the View."

You and the Document

Table 4.2 shows your responsibilities and those of the framework in implementing a document.

Table 4.2 Document Implementation Responsibilities

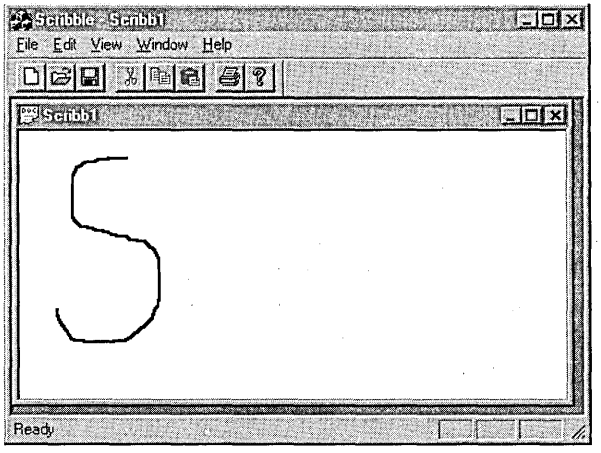
Your job	The framework's job
Derive a document class from class **CDocument**.	Provide many document services through class **CDocument**.
Add data members to your class.	
Implement application-specific initialization and cleanup of your document's data.	Call the appropriate initialization and cleanup functions at the right times.
Override **CDocument**'s **Serialize** member function to specify how your data is read and written.	Provide implementations of File Open, Save, and Save As that call your `Serialize` override to read and write your data.

Typically, you also add member functions to your derived document class through which other objects—mainly the view—can access the document's data.

Scribble's Document: Class CScribbleDoc

Scribble is a simple drawing program. Documents in Scribble store the lines, or "strokes," that make up a drawing. Because a drawing is typically made up of many strokes, the document stores a list of all the strokes the user has drawn. Figure 4.4 shows a single stroke drawn in a Scribble document's view.

Figure 4.4 One Stroke in Scribble



Documents in Scribble are objects of class `CScribbleDoc`, which is derived from **CDocument**. `CScribbleDoc` has a member variable named `m_strokeList` for storing a list of strokes and member functions to manage the stroke list. You can view this member variable in the Classes pane by expanding the icon for class `CScribbleDoc`.

Note By convention, class names begin with an uppercase "C" and member variable names begin with a lowercase "m_".

Customizing CScribbleDoc

AppWizard writes a skeletal `CScribbleDoc` class for you, which you'll customize in the following procedures. First you'll add `CStroke`, a class used to define the document's data structure, to `CScribbleDoc`. Then you'll add member variables and functions to `CScribbleDoc`, which provide some typical document functionality, such as:

- Defining and manipulating the document's data.
- Serializing the data to and from files.

In this chapter, and throughout this tutorial, you'll use the text editor to add code to an existing file (one of the files AppWizard created for you).

Jumping Directly to Code

Microsoft Developer Studio provides several tools that make navigating through your code quick and easy. ClassView visually depicts all existing classes and their members. WizardBar, which appears at the top of the text editor window when you have an implementation (.cpp) file open, lists all virtual functions, Windows messages, and **CCmndTarget** procedures associated with the selected class.

You can use either of these tools to jump directly to the code you'll edit. For more information, see "Navigating Through Code," for a description of some basic procedures you can use to jump to your code.

Adding Code to CScribbleDoc

In the following procedure you add the forward declaration for `CStroke`. You'll learn more about `CStroke` later in this chapter, in the section "The Document's Data: Class CStroke." Then, in procedures that follow, you'll add new member variables (`m_nPenWidth` and `m_penCur`) and functions (`InitDocument` and `NewStroke`) to `CScribbleDoc`. For more information, see "Adding Members from ClassView" in Chapter 2, of the *Visual C++ User's Guide*.

Note The source files for tutorials such as Scribble contain comments that clarify the purpose of the code. Including such comments is not always mentioned as part of a tutorial step, both for brevity's sake, and because the the purpose of code you add is explained in the procedures you'll follow. You can always, of course, modify or add to comments in your own files.

▶ To add CStroke to CScribbleDoc,

1 In ClassView, expand the project folder, if necessary, and double-click the icon for the `CScribbleDoc` class.

The document class header file, ScribbleDoc.h, opens in the text editor with the cursor placed at the class definition.

2 Add the following code just before the definition of class `CScribbleDoc:`

```
// Forward declaration of data structure class
class CStroke;
```

3 Save the header file. (Optional.)

Note Developer Studio saves all your work for you automatically when you build your project, close the workspace, or exit the program. Any prompts within these procedures to save your files are provided only as extra safeguards of your work.

▶ To add member variables to Scribble's document class

1 In ClassView, point the cursor at `CScribbleDoc` and click the right mouse button.

2 In the pop-up menu, choose Add Variable.

The Add Member Variable dialog appears.

3 From the Variable Type drop-box, select UINT (or simply type it in).

4 In the Variable Declaration box, type `m_nPenWidth`.

5 In the Access area, select Protected, and choose OK.

Class Wizard adds the variable declaration to file ScribbleDoc.h.

6 Repeat steps 1 through 5 to declare the protected `m_penCur` variable of type **CPen**.

ClassView displays the new member declarations and variables you've added with your code.

7 Expand the icon for `CScribbleDoc`.

Note that `m_nPenWidth` and `m_penCur` are now displayed as variables belonging to the class. Once member functions and variables are declared, ClassView displays them, even if their definition isn't written yet.

8 Double-click the icon for `m_nPenWidth`.

ScribbleDoc.h opens in the text editor, with your cursor at the definition for `m_nPenWidth`.

9 Add the following two lines of code in the Public attributes section of ScribbleDoc.h:

```
CTypedPtrList<CObList, CStroke*> m_strokeList;
CPen*   GetCurrentPen( ) { return &m_penCur; }
```

The `m_strokeList` variable is an instance of a template. The `GetCurrentPen` function is inline, so its definition appears in ScribbleDoc.h instead of ScribbleDoc.cpp.

▶ To add member functions to Scribble's document class

1 In ClassView, point the cursor at `CScribbleDoc` and click the right mouse button.

2 In the pop-up menu, choose Add Function.

The Add Member Function dialog appears.

3 Fill in the dialog as follows:

- In the Function Type box, type `CStroke*`.
- In the Function Declaration box, type `NewStroke()`.
- In the Access area, select Public.
- Choose OK.

A declaration is added to the file ScribbleDoc.h, while a skeletal definition is added to ScribbleDoc.cpp. You'll fill in the definition later in this chapter.

4 Repeat steps 1 through 3 to declare the function `InitDocument` with a return type of **void**, and a Protected access specifier.

Once again, you can view the new functions in ClassView.

Adding the Template Collection Classes to Scribble

Scribble uses two of several C++ collection template classes provided by MFC: **CTypedPtrList** and **CArray**. You've just added the code that uses **CTypedPtrList**; you'll add the code that uses **CArray** when you declare the **CStroke** class.

All the template collection classes are defined in the header file, AFXTEMPL.H. Since this MFC-provided header file will not change over the course of your development of the Scribble application, it makes sense to add it to Scribble's precompiled header, STDAFX.H.

STDAFX.H was created by AppWizard to keep the list of header files to be precompiled. It consists of a list of **#include** statements, followed by the names of the header files.

Note Precompiled header (PCH) files speed up build times because they don't require recompiling. For more information, see "Precompiled Headers."

▶ To add AFXTEMPL.H to the precompiled header

1 Switch to FileView, and if necessary, expand the Scribble project folder.

2 Expand the Dependencies folder, and open file StdAfx.h.

3 At the bottom of the file, type

```
#include <afxtempl.h>    // MFC templates
```

4 Save StdAfx.h.

Using WizardBar to Override Base-Class Functions

The final task in declaring the member functions for `CScribbleDoc` is to override **OnOpenDocument** and **DeleteContents**, which are needed for Scribble-specific initialization and clean-up.

You may be familiar with how to use ClassWizard to override virtual functions. Using the WizardBar (a short cut into ClassWizard) eliminates several steps for you. For more information, see "Using WizardBar," in Chapter 14, "Working with Classes," of the *Visual C++ User's Guide*.

Note The following procedures assume you have an editor window open, and that it is displaying the implementation (.cpp) file containing the class whose functions you want to override. By default, WizardBar should be displayed. If it is not, first use the following procedure to display it.

▶ To display the WizardBar

- With your cursor in the editor window, click the right mouse button, and from the pop-up menu, choose Toolbar.

  The Toolbar command is a toggle switch that either displays or hides the WizardBar.

▶ To use the WizardBar to override functions

1 If necessary, open the class implementation file, in this case, ScribbleDoc.cpp.

  In the WizardBar Object IDs List, the class name (in this case `CScribbleDoc`) is selected by default.

2 From the Messages drop-down list, select the function you want to override. First select OnOpenDocument.

   A message box prompts you to specify whether you want to add a handler for the function.

3 Choose Yes.

   ClassWizard creates a skeleton definition for the function, highlighting the code so you can simply begin typing to override it. (You won't type anything at this point.)

4 Repeat steps 2 and 3 for DeleteContents.

5 Save ScribbleDoc.h and ScribbleDoc.cpp.

ClassWizard writes the overridden functions to ScribbleDoc.h. As a result, you can view them in ClassView.

Here's how the ClassWizard-generated code appears in ScribbleDoc.h:

```
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CScribbleDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
    virtual void DeleteContents();
    //}}AFX_VIRTUAL
```

AppWizard added the implementation for `OnNewDocument`, and for `Serialize`, to ScribbleDoc.h when you first created the skeleton application. (You may have noticed that both functions already appeared in bold in the WizardBar Messages list.)

Here is the skeleton function handler code that ClassWizard added to ScribbleDoc.cpp:

```
BOOL CScribbleDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;

    // TODO: Add your specialized creation code here

    return TRUE;
}

void CScribbleDoc::DeleteContents()
{
    // TODO: Add your specialized code here and/or call the base
class

    CDocument::DeleteContents();
}
```

You will fill in the handler definitions for `OnOpenDocument`, `OnNewDocument`, and `DeleteContents` later, in "Initializing and Cleaning Up."

Summary of CScribbleDoc Member Functions and Variables

Table 4.3 describes the current member variables of class `CScribbleDoc`, and the purpose they will serve in Scribble.

Table 4.3 CScribbleDoc Data Members

Member	Description
`m_strokeList`	A list of strokes. Each item in the list is an object of class `CStroke`. The list itself is a C++ class template based on the MFC template class, **CTypedPtrArray**.
`m_penCur`	A **CPen** object used to do the drawing. Its main attribute is its width. The pen is created when the document is constructed and is used during the creation of new strokes.
`m_nPenWidth`	The current width of the lines drawn by the pen.

Table 4.4 describes the member functions.

Table 4.4 CScribbleDoc Member Functions

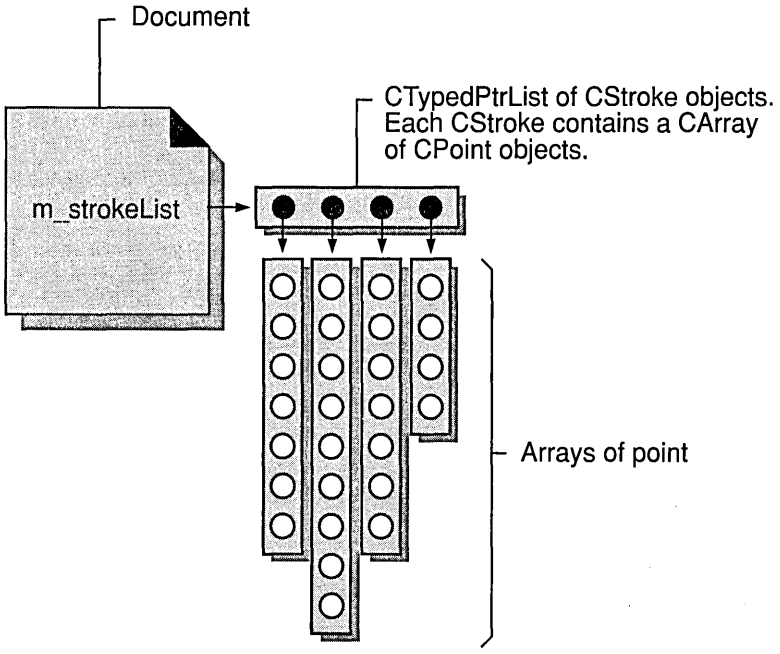
Member	Description
`CScribbleDoc`, `~CScribbleDoc`	A default constructor and a virtual destructor. AppWizard creates placeholders for these functions. In Scribble, they remain empty.
`DeleteContents`	Deletes the contents of a document—the strokes that make up the drawing.
`GetCurrentPen`	Retrieves a pointer to the current pen object any time it's needed by the drawing code.
`InitDocument`, `OnNewDocument`, `OnOpenDocument`	Called when a new document is created or an existing document is opened. The overridden versions of the **CDocument** member functions **OnNewDocument** and **OnOpenDocument** call `InitDocument` to initialize the new document.
`NewStroke`	Creates a new stroke object and adds it to the list of strokes in `m_strokeList`.
`Serialize`	Overrides the **Serialize** member function of class **CDocument**. The override specifies how to serialize a list of stroke objects to and from a disk file. AppWizard creates this function for you in skeletal form.
`AssertValid`	Tests the validity of an object's internal state.
`Dump`	Dumps the contents of an object's members for examination during debugging.

You'll add code for most of these member functions in later sections of this chapter. You'll learn more about `Serialize` under "Serializing the Data." For more information about **AssertValid** and **Dump**, see "Diagnostics" in *Programming with MFC*. You won't add code to these functions for Scribble.

The Document's Data: Class CStroke

In Scribble, a stroke consists of an array of points. As the user drags the mouse to draw, Scribble collects points and stores them as part of the current stroke. Points collected from the time the left mouse button is pressed to the time it's released form one stroke of a Scribble drawing. Figure 4.5 shows Scribble's data structure schematically. Scribble uses an object of class **CPen** for drawing.

Figure 4.5 Scribble's m_strokeList Data Structure



Each stroke is stored in an object of class `CStroke`, Scribble's primary data structure, which you'll declare as a new class in Scribble's source files (ScribbleDoc.h and ScribbleDoc.cpp). The whole drawing is a list of `CStroke` objects.

The next topic describes how to add CStroke to Scribble.

Adding the CStroke Class

Recall that you added a forward declaration of `CStroke` previously. Now you will add the actual `CStroke` class declaration.

▶ To add the CStroke class

- If you're working along, add the code shown below to ScribbleDoc.h, and then save the file. The declaration for class `CStroke` follows that for class `CScribbleDoc`, so it comes at the very end of the file.

```
// class Cstroke
// A stroke is a series of connected points in the Scribble drawing.
// A Scribble document may have multiple strokes.
class CStroke : public CObject
{
public:
    CStroke( UINT nPenWidth );

protected:
    CStroke( );
    DECLARE_SERIAL( CStroke )

// Attributes
protected:
    UINT   m_nPenWidth;  // One width applies to entire stroke
public:
    CArray<CPoint, CPoint>   m_pointArray;  // Series of connected
                                            // points

// Operations
public:
    BOOL DrawStroke( CDC* pDC );

public:
    virtual void Serialize( CArchive& ar );
};
```

This code declares a C++ class of stroke objects. You can examine the new `CStroke` class and its current list of member functions and variables in ClassView. The next topic, "CStroke Member Functions and Variables," describes their purpose.

Your next step in this tutorial is to add definitions for `CStroke`'s member functions.

CStroke Member Functions and Variables

The CStroke class member variables and functions will be used to define and manipulate the data of a stroke and serialize it when the document is serialized. You'll add the member function definitions in the next section, "Building and Storing Strokes."

Table 4.5 lists CStroke's member variables.

Table 4.5 CStroke Data Members

Member	Used to
m_nPenWidth	Store the width of the pen in effect at the time this stroke was drawn.
m_pointArray	Store an array containing the points that define this stroke. The points are used to redraw the stroke as needed.

Table 4.6 lists CStroke's member functions.

Table 4.6 CStroke Member Functions

Member	Used to
CStroke	The class defines two constructors: one protected and one public.
DrawStroke	Draws each stroke. When the view object redraws the document's data, it calls upon each stroke object in the stroke list to draw itself by calling its DrawStroke member function.
Serialize	Assist the document in making its data persistent, typically on disk. CStroke overrides the **Serialize** member function of **CObject** to define how a single stroke serializes its points and other data. For more information about point serialization, see "Serializing the Data."

Building and Storing Strokes

In this tutorial step you add definitions for CStroke's member functions.

▶ To add implementation code for the CStroke members

- Add the following definitions for CStroke's two empty constructors to ScribbleDoc.cpp, after the last line of CScribbleDoc code:

```
/////////////////////////////
// CStroke
CStroke::CStroke()
{
```

```
// This empty constructor should be used by serialization only
}

CStroke::CStroke(UINT nPenWidth)
{
    m_nPenWidth = nPenWidth;
}
```

The first constructor, declared protected in ScribbleDoc.h, is used only by the application framework during serialization of `CStroke` objects. Its parameter list and function body are empty. The second constructor is for public use, when you need to construct new stroke objects directly. When it constructs a new stroke object, the public constructor initializes the pen width. `CStroke` doesn't declare its own destructor—it relies on **CObject** to provide one by default.

At this point, class `CStroke` is not quite complete. You'll add code for the remaining member function, `DrawStroke`, in "Constructing the User Interface." This member function is used by the view object to draw the data.

Managing the Document

Typically, you must write code to:

- Initialize a document's data members
- Deallocate memory allocated for the data, release system resources, and perform other cleanup chores.

When a new Scribble document is created, `CScribbleDoc` must create a pen for drawing new strokes. When a document is closed, the document must delete the stroke objects it has stored up.

The following section, "Initializing and Cleaning Up," describes how to do this for Scribble.

Initializing and Cleaning Up

Because a document can be created with either the New command or the Open command on the File menu, `CScribbleDoc` overrides both the **OnNewDocument** and **OnOpenDocument** member functions of **CDocument** to perform necessary document initialization. However, for Scribble, both initializations are the same, so both overrides call the new member function `InitDocument`. (Recall when you declared `InitDocument` earlier in this chapter.)

The framework automatically calls `OnNewDocument` when a new document is created or `OnOpenDocument` when a document is opened. AppWizard creates a skeletal version of `OnNewDocument` for you, and ClassWizard created a skeleton version for `OnOpenDocument`.

In the following procedure, you'll define these functions for Scribble.

▶ To implement initialization for Scribble's documents

1 Use ClassView to jump to the skeletal definition for `InitDocument` (in class `CScribbleDoc`), and fill it in with the following code:

```
m_nPenWidth = 2;  // Default 2-pixel pen width
// Solid black pen
m_penCur.CreatePen( PS_SOLID, m_nPenWidth, RGB(0,0,0) );
```

`InitDocument` sets a default pen width and creates a pen object for drawing. Pen creation is done through the **CPen** object, `m_penCur`, by calling its **CreatePen** member function. The arguments specify a solid black pen 2 pixels wide.

2 Use ClassView or WizardBar to jump to the override of `OnNewDocument` created by AppWizard, and replace the \\TODO comments with the call to `InitDocument`. The completed handler looks like this:

```
BOOL CScribbleDoc::OnNewDocument()
{
    if(!CDocument::OnNewDocument())
        return FALSE;
    InitDocument();
    return TRUE;
}
```

3 Finally, jump to the skeletal version of `OnOpenDocument`, and replace the \\TODO comments with another call to `InitDocument`. The completed handler looks like this:

```
BOOL CScribbleDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if( !CDocument::OnOpenDocument( lpszPathName ) )
        return FALSE;
    InitDocument( );
    return TRUE;
}
```

The overrides of `OnNewDocument` and `OnOpenDocument` call the base-class version of the function before performing application-specific initialization of the document.

The following procedure describes how to add the code that implements document cleanup for Scribble. You'll replace the ClassWizard skeleton version of the **DeleteContents** member function of **CDocument.** Note that you are still working in ScribbleDoc.cpp.

▶ To implement document cleanup

- Use ClassView or WizardBar to jump to the `DeleteContents` member function, and replace the //TODO comment with the following code:

```
while( !m_strokeList.IsEmpty( ) )
{
delete m_strokeList.RemoveHead( );
}
<<<
```

`DeleteContents` provides the best place to destroy a document's data when you want to keep the document object around. The function is called automatically by the framework any time it's necessary to delete only the document's contents. It's called in response to the Close command on the File menu, when the user closes the document's last open window, and before creating or opening a document with the New and Open commands. This is all part of the base-class functionality of `DeleteContents`.

Scribble's override of `DeleteContents` iterates through the stroke list. For each stroke object, the function invokes the **delete** operator. This destroys the strokes. **RemoveHead**, a member function of class **CTypedPtrList**, does two things: it removes the pointer to the object from the head of the list, and it returns the pointer. Then it deletes the object that the pointer points to.

Alternatively, this cleanup code could be placed in the destructor, but `DeleteContents` is reused later in other functions.

Managing the Data

In this procedure, you'll create a member function, `NewStroke`, that manages the stroke data in Scribble's drawings.

`NewStroke` uses the C++ **new** operator to construct a new `CStroke` object dynamically, initializing it with the current pen width. It uses the **CTypedPtrList** member function **AddTail** to add the new stroke to the list. Then it calls the **CDocument** member function **SetModifiedFlag** to flag the document as having been modified since the last save, and it returns a pointer to the stroke.

Just as you implemented the skeleton handler for `InitDocument`, in the following procedure you'll implement the `NewStroke` function that you declared earlier. `NewStroke` creates a new stroke object and adds it to the stroke list.

▶ To implement document members for managing Scribble's data

- Use ClassView to jump to the skeletal definition of `NewStroke` (in class `CScribbleDoc`), and fill it in with the following code:

```
CStroke* pStrokeItem = new CStroke(m_nPenWidth);
m_strokeList.AddTail( pStrokeItem );
SetModifiedFlag( ); // Mark document as modified
                    // to confirm File Close.
return pStrokeItem;
```

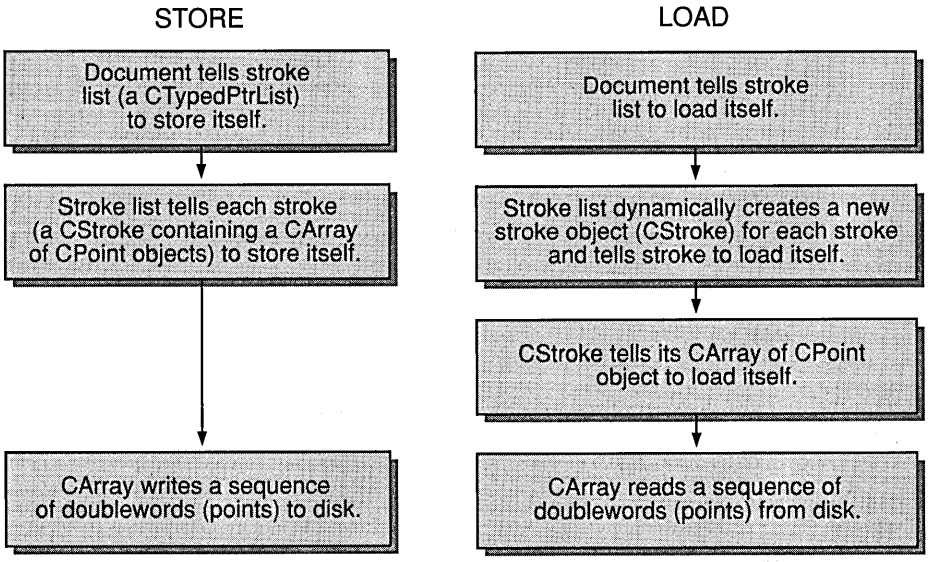
Note that the **new** operator never returns NULL. Instead, an exception is thrown if memory could not be allocated. This would be a good place to implement an exception handler with the **TRY** and **CATCH** macros. For more information about exception handling, see the article "Exceptions" in *Programming with MFC.*

Serializing the Data

This section describes how to add the code that defines file input/output for Scribble documents. The default I/O implementation in MFC is called "serialization" (see Figure 4.6). It provides a mechanism for making a document's data persistent between work sessions with the program. By adding the code described in this section, you enable the Scribble application to handle file serialization when the user chooses the Open, Save, or Save As commands from the File menu.

Note Other than filling in the serialization function, you don't have to write any code—for example, to display the dialog boxes—to process the Open, Save, and Save As commands on the File menu. The framework supplies this code.

Figure 4.6 Serialization in Scribble



The `CScribbleDoc` class declaration in file ScribbleDoc.h begins with the following lines, which contain an important macro invocation (**DECLARE_DYNCREATE**) needed for serialization (don't add this code):

```
class CScribbleDoc : public CDocument
{
protected: // Create from serialization only.
    CScribbleDoc( );
    DECLARE_DYNCREATE( CScribbleDoc )
    // Other declarations ...
};
```

AppWizard wrote this code for you when you first created the skeleton application.

The **DECLARE_DYNCREATE** macro prepares the class so that document objects can be dynamically created by the framework.

Serializing the Document

Serializing a document occurs in two stages. First, the framework calls the document's `Serialize` member function. Second, that `Serialize` function calls the **Serialize** function of the stroke list.

In the following procedure, you'll add code that implements serialization for Scribble's documents.

▶ To implement serialization for Scribble documents

- Use ClassView or WizardBar to jump to the `Serialize` member function in `CScribbleDoc`, and add the following line just before the closing brace:

  ```
  m_strokeList.Serialize( ar );
  ```

  Later, you'll add code to both branches of the **if** statement.

  Serialization uses an object of class **CArchive** to manage the connection to a disk file or other storage. A **CArchive** object, **ar**, is passed in as an argument.

  A call to the archive object's **IsStoring** member function determines whether this is a store or a load operation. If the archive is for storing (saving), the stroke-list object's own `Serialize` member function is called to store the stroke's data to disk. If the archive is for loading, its `Serialize` member function is called to load data from the disk file. This constructs new `CStroke` objects to fill the list. The stroke list for a document being read in from disk must already be empty.

  Note that the stroke list already exists when `Serialize` reads data in. That's because it was declared as an embedded object, like this:

  ```
  CTypedPtrList <CObList, CStroke*> m_strokeList;
  ```

  rather than as a pointer, like this:

  ```
  CTypedPtrList <CObList, CStroke*>* m_pStrokeList;
  ```

  For a pointer, you'd use **CArchive**'s extraction (>>) operator to read the data:

  ```
  ar >> m_pStrokeList; // Example of serializing to a
                       // referenced (non-embedded) object
  ```

  But for an embedded object, as in Scribble, you call `Serialize` directly because you don't want to create a second **CTypedPtrList** object, and because you know the exact type of the object.

Serializing Strokes

When the document responds to an Open, Save, or Save As command, it delegates the real serialization work to the strokes themselves. That is, the document tells the stroke list to serialize itself, and the stroke list, in turn, tells the individual strokes to serialize themselves. As a result, all strokes in the document are read from or written to a file.

Note Throughout the tutorial, Scribble is presented as a series of incremental versions. When you build successive versions that modify the structure of `CStroke`, they are incompatible with earlier versions. Attempts to read `CStroke` data stored by a previous version may fail because the serialization process expects a different structure. Each time you make such a modification of `CStroke`, it's valuable to tag the new version with a version number. The version or "schema" number is checked automatically during serialization. You can check the schema number in the serialization code to support backward compatibility, allowing you to read files created with earlier versions of your application.

For more information, see **CArchive::GetObjectSchema** in the *Class Library Reference*.

You've added serialization for Scribble documents; now you'll implement serialization for the strokes.

▶ To implement serialization for stroke objects

1 Use ClassView to jump to the `CStroke` constructor and, just before it, add the **IMPLEMENT_SERIAL** macro as shown:

```
IMPLEMENT_SERIAL( CStroke, CObject, 1 )
```

The third argument is the schema number, discussed earlier. It's set to 1 for Scribble Step 1.

The **IMPLEMENT_SERIAL** macro complements the **DECLARE_SERIAL** macro which you declared in ScribbleDoc.h when you were adding the code for the `CStroke` class. The two macros prepare a class for serialization.

Like `CScribbleDoc`, `CStroke` also overrides the **Serialize** member function of its base class. When the stroke-list object is called to serialize itself, it calls each stroke object in turn to serialize itself.

2 Add the following `Serialize` override for class `CStroke`. This code should come just after the second `CStroke` constructor (the one that initializes the pen).

```
void CStroke::Serialize( CArchive& ar )
{
    if( ar.IsStoring( ) )
    {
        ar << (WORD)m_nPenWidth;
        m_pointArray.Serialize( ar );
    }
    else
    {
        WORD w;
    ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize( ar );
    }
}
```

If the archive object is used for storing, the stroke's pen-width value is stored in the archive and then its array of points is stored. Notice that the `m_pointArray` object as a **CArray** object can serialize itself.

If the archive object is used for loading, the stroke's data must be read in the same order it was written: first the pen width, then the array of points. The **else** branch of the **if** statement declares a local variable to receive the width, then copies that value to `m_nPenWidth`. It then calls upon the point array to load its data (see Figure 4.6).

Note that the `m_nPenWidth` variable is cast to a **WORD** before it's inserted in the archive, **ar**:

```
ar << (WORD)m_nPenWidth;
```

The cast is necessary because `m_nPenWidth` is declared as type **UINT** (unsigned integer). The archive mechanism only supports saving types of fixed size. **UINT**, for example, is 16 bits in Windows 3.1 and 32 bits in Windows NT and Windows 95. Using the **WORD** cast makes the data files created by your application portable. To promote machine independence, class **CArchive** doesn't have an extraction operator for type **int** but does have one for type **WORD**.

Once this code is in place, serialization of Scribble's data is automatic.

Creating the Document: Summary

In this chapter you filled in the details of Scribble's document class by defining its data, providing useful functions through which to manipulate the data, and specifying how the data objects are written to and read from files. So far, the data can be initialized and cleaned up but not displayed or worked on by the user.

At this point, Scribble is about half ready to build. In Chapter 5, "Creating the View," you'll complete the basic Scribble application by developing a view on the document. The view displays strokes and manages all user input. At the end of that chapter, you'll build and test Scribble.

CHAPTER 5

Creating the View

In Chapter 4, "Creating the Document," you completed Scribble's document class by adding code to the ScribbleDoc.h and ScribbleDoc.cpp files. In this chapter, you'll add a view class that provides a "view on the document." To do so, you'll add code to Scribble's view files, ScribbleView.h and ScribbleView.cpp. You'll also add two more member functions to class `CStroke` in ScribbleDoc.cpp.

By the end of this chapter, you can compile and run Scribble.

Scribble's view class displays the strokes of a drawing and accepts user input from the mouse. Among the things you'll develop in this chapter are:

- Code to display Scribble's strokes— in class `CScribbleView`.
- Code to handle Windows messages as the user draws with the mouse — also in class `CScribbleView`.

While you're adding code, you'll have more opportunities to use ClassView to jump directly to your code. You'll also get more hands-on experience with WizardBar. WizardBar, a shortcut into ClassWizard, lets you map Windows messages to message-handler member functions in your classes. As you'll see in Chapter 7, "Binding Visual Objects to Code Using WizardBar," it also lets you map the commands generated by user-interface objects such as menu items, toolbar buttons, and accelerator keys to message-handler functions.

This chapter and Chapter 4 cover Step 1 of Scribble. If you want to work along, adding the code as you go, begin with the files from your Scribble root project directory. At this point, your files should consist of the starter files you created with AppWizard in Chapter 3 and modified in Chapter 4. As you read this chapter, add all lines of code as instructed in the procedures. At the end of this chapter, your files should closely resemble the Step 1 source code files.

If, on the other hand, you want to read along without adding code, you can print or examine the Step 1 files in Books Online.

Note If you have not made a local copy of the sample source code for the tutorial step you're working in, it's easy to do so. For more information, see "Installing the Sample Files" in Chapter 2.

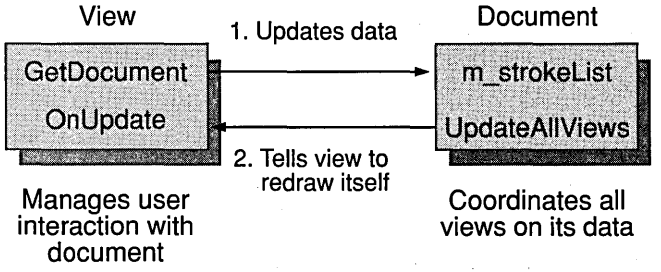
Views

The document object defines, stores, and manages the application's data. All user interaction with the document, however, is managed through a view object attached to the document object. Scribble uses a view object to display a document on the screen or on a printer. This section explains the role of the view and introduces Scribble's view class and its members.

As you saw in Chapter 4, when a new document is created in response to a New or Open command from the File menu, the framework also creates a document frame window and creates a view inside the frame window's client area as a child window. The view displays the document's data and responds to mouse actions, keystrokes, menu commands, and other actions as the user works on the document. It's your task to specify how the view draws your application-specific data and what it does in response to user actions.

Figure 5.1 illustrates the view's role in relation to the document.

Figure 5.1 The View and the Document



View Definition

A view is an object derived from class **CView** (or from another **CView**-derived class, such as **CScrollView**) that manages user interaction with a document. The view is a child window that typically fills the client area of a document frame window. In single document interface (SDI) applications, the view fills the main frame window. In multiple document interface (MDI) applications, the view fills the MDI child.

Views in the Framework

In the framework, the document manages data, but the view displays it and acts as intermediary between the user and the document for all input, selection, and editing in the document. A given view is always associated with only one document. However, it is possible for a document to have multiple views associated with it. In MDI applications such as Scribble, for example, the Window menu contains commands to open a new view onto a document, and to arrange open documents. In Scribble, you'll also add splitter window functionality, enabling the user to create a second view onto a document without opening a new window.

For more information about documents and views, see Chapter 9, "Enhancing Views."

View Creation

A view is created by its parent frame window when the framework creates the associated document. Both the document and frame objects are created by a document template object; then the frame window creates the view. Immediately after creation, the framework calls the view's **OnInitialUpdate** member function to initialize the view. You'll frequently override the **OnInitialUpdate** member function of class **CView** to initialize the view object. After creation, the view's **OnUpdate** member function is called when the document's data changes. You can override the **OnUpdate** member function to optimize which portion of the view is redrawn.

Drawing the View's Contents

Each time the view needs to be redrawn, the framework calls its **OnDraw** member function. **OnDraw** does the actual drawing, obtaining the data to draw from its document. However, when more immediate drawing is required, a view can respond to mouse-related messages, such as **WM_LBUTTONDOWN**, to do mouse-driven drawing. You'll see both kinds of drawing in this chapter.

You'll always override the **OnDraw** member function of class **CView** to specify how your document's data is drawn.

How the View and Document Interact

A view can access the data stored in its document by calling the **CView** member function **GetDocument,** which returns a pointer to the document object. The view can call public member functions and access public data members of the document by using the pointer.

When the user changes data in the view, the view notifies the document and updates the data stored there. On such occasions, the document typically then calls its **UpdateAllViews** member function to cause any views attached to it to redraw themselves. For a document with multiple views, this mechanism ensures that all of them are updated properly.

You and the View

Table 5.1 shows your responsibilities and those of the framework in implementing a view on a document.

Table 5.1 View Implementation Responsibilities

Your job	The framework's job
Derive a view class from class **CView**. For scrolling, use **CScrollView** instead. Other view classes are available as well.	AppWizard provides a skeletal view class for you. Class **CView** and its derived classes provide view services.
Implement your view's `OnDraw` member function.	The framework calls `OnDraw` at the appropriate times, passing it a device-context object into which it can draw.
Map Windows messages and commands to member functions of your view.	The framework calls your message-handler member functions in response to the corresponding Windows messages.

Other view classes include, for example, **CFormView**, **CEditView**, **CListView**, **CScrollView** and **CTreeView**. To see the complete list of **CView**-derived classes, see class **CView** in the *Class Library Reference*. For more information about views, see Chapter 1, "Using the Classes to Write Applications for Windows," and Chapter 3, "Working with Frame Windows, Documents, and Views," of *Programming with MFC*

Scribble's View: Class CScribbleView

The job of the view in Scribble is to redraw the view as needed—for example, when the window is covered by another window and then uncovered, or as the user draws strokes with the mouse.

Views in Scribble are objects of class `CScribbleView`, which is derived from class **CView**. `CScribbleView` knows how to access the document's stroke list and can tell the strokes stored there to draw themselves in the view.

You can view a graphical representation of `CScribbleView` and all its member functions in the ClassView pane of the Project Workspace window.

Table 5.2 describes the member functions of class `CScribbleView` that AppWizard created for you.

Table 5.2 CScribbleView Member Functions

Member	Description
`CScribbleView`, `~CScribbleView`	With nothing to initialize and no data to destroy, the view's constructor and destructor are empty.
`OnDraw`	`OnDraw` updates the view by redrawing its contents. (It's used to draw both on the screen and on a printer.)
`GetDocument`	Defined inline in file ScribbleView.h, `GetDocument` retrieves a type-safe pointer to the document attached to this view. The view uses the pointer to call document member functions, which it must do to access the data it displays.
`AssertValid`, `Dump`	These diagnostic functions simply call the base-class functions they override.
`OnPreparePrinting`, `OnBeginPrinting`, `OnEndPrinting`	These virtual functions override the versions in **CView** to specify the application's printing behavior. See Chapter 10, "Enhancing Printing," for more information about how Scribble prints.

You won't need to alter any of the following AppWizard-created functions in this chapter: `GetDocument`, `AssertValid`, `Dump`, `OnPreparePrinting`, `OnBeginPrinting`, and `OnEndPrinting`.

Notice the inline definition of `GetDocument` after the class declaration above. The debug version of this member function calls the **IsKindOf** member function defined in class **CObject** and uses the **RUNTIME_CLASS** macro to retrieve the run-time class name of the document. For more information about those topics, see class **CObject** in the *Class Library Reference* and the article "CObject Class" in *Programming with MFC*.

The next sections describe the code you'll add to `CScribbleView`.

Defining the Working Data Used by the View

The view's job in Scribble is to redraw itself as needed — for example, when the window is covered by another window and then uncovered, or as the user draws with the mouse.

In the procedure in this section, you'll add two member variables to class `CScribView` that store information about a stroke in progress.

Note The procedures in this chapter assume you are familiar with the various tools, such as ClassView and WizardBar, that Microsoft Developer Studio provides to make working with your project code easy and intuitive. For more information, see "Navigating Through Code" in Chapter 3, "Creating a New Application with AppWizard."

▶ To declare the new member variables

1 In ClassView, double-click the icon for the `CScribbleView` class.

This jumps you directly to the class definition (generated for you by AppWizard) in file ScribbleView.h.

2 Add the following code right after the public Attributes section:

```
protected:
    CStroke*   m_pStrokeCur; // The stroke in progress
    CPoint     m_ptPrev;     // The last mouse pt in the stroke
                             // in progress
```

The New CScribbleView Member Variables

The code you just added declares two new protected member variables inside class `CScribbleView` — `m_pStrokeCur` and `m_ptPrev`.

You can view the new member variables in the ClassView pane of the Project Workspace window by expanding the `CScribbleView` class icon.

Table 5.3 describes the new member variables.

Table 5.3 CScribbleView Data Members

Member	Description
`m_pStrokeCur`	A pointer to the stroke currently being drawn.
`m_ptPrev`	A **CPoint** object containing the previous mouse coordinates, from which a line will be drawn to the current coordinates.

The view uses these members to store the information it needs in order to record the points of a stroke in progress.

Redrawing the View

When the view, or some part of it, must be redrawn, the framework calls the override of the `OnDraw` member function that AppWizard generated. In this section you'll add Scribble-specific code for `OnDraw` to the ScribbleView.cpp file.

▶ To add implementation code for the view's OnDraw member function

1 Use ClassView or WizardBar to jump to the skeleton `OnDraw` member function of class `CScribbleView`.

(You can use WizardBar to jump to functions when the .cpp file is open in the current editor window. Using ClassView automatically opens the file for you.)

Your cursor is placed inside the skeleton code that AppWizard created in ScribbleView.cpp for the `OnDraw` function.

2 Add the following code after the `ASSERT_VALID(pDoc)` line. (You can replace the // TODO comments for adding drawing code.)

```
// The view delegates the drawing of individual strokes to
// CStroke::DrawStroke( ).
CTypedPtrList<CObList, CStroke*>& strokeList =
        pDoc->m_strokeList;
POSITION pos = strokeList.GetHeadPosition( );
while (pos != NULL)
{
    CStroke* pStroke = strokeList.GetNext(pos);
    pStroke->DrawStroke( pDC );
}
```

The view calls upon the individual stroke objects to draw themselves. To do this, the view needs access to the stroke data stored in the document, so the view's first task is to obtain a pointer to its document, using `GetDocument`. The view then uses the pointer to iterate through the stroke list, telling each stroke to draw itself. When `OnDraw` calls `DrawStroke` for a given stroke object, it passes along the device-context object it received as a parameter. (Having the data draw itself is only one possible strategy.)

To complete Scribble's drawing, you must also add the `DrawStroke` member function definition to class `CStroke`.

▶ To add drawing code for strokes

- Add the `DrawStroke` member function definition to ScribbleDoc.cpp as shown, right after the Serialize function. (Recall that you added its declaration when you added the `CStroke` class to ScribbleDoc.h.)

```
BOOL CStroke::DrawStroke( CDC* pDC )
{
    CPen penStroke;
    if( !penStroke.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)))
        return FALSE;
    CPen* pOldPen = pDC->SelectObject( &penStroke );
    pDC->MoveTo( m_pointArray[0] );
    for( int i=1; i < m_pointArray.GetSize(); i++ )
    {
        pDC->LineTo( m_pointArray[i] );
    }
    pDC->SelectObject( pOldPen );
    return TRUE;
}
```

This code passes `DrawStroke` a pointer to an object of class **CDC,** which encapsulates a Windows device context (DC). In programs written with MFC, all graphics calls are made through a device-context object of class **CDC** or one of its derived classes. `DrawStroke` calls **CDC** member functions—**SelectObject**, **MoveTo**, **LineTo**—through the pointer to select a graphic device interface (GDI) pen into the device context and to move the pen and draw.

`DrawStroke` next constructs a new **CPen** object and initializes it with the current properties by calling the pen's **CreatePen** member function—note that this two-stage construction is typical of framework objects. Then `DrawStroke` calls **SelectObject** to select the pen into the device context (saving the existing pen as `pOldPen`) and calls **MoveTo** to position the pen to the first point.

`DrawStroke` then iterates through the array of points. It calls the device context's **LineTo** member function to connect the previous point with the next point.

Finally, `DrawStroke` restores the device context to its previous condition by reinstalling its old pen.

Important Always restore the device context to its original state before releasing it to Windows. To do so, save the state before you change it. Storing the old pen in `DrawStroke` is an example of how to do this.

The addition of `DrawStroke` completes Scribble's code for drawing in response to update requests from the framework. However, Scribble also draws in response to mouse actions, as discussed in the next section, "Handling Windows Messages in the View."

Handling Windows Messages in the View

To implement mouse-driven drawing in Scribble, it's necessary to write code that handles several Windows messages related to mouse activity. You will use the WizardBar to help write this message-handling code.

When the user presses the left mouse button while the pointer is in a Scribble window, Windows sends the window a **WM_LBUTTONDOWN** message. If the user moves the mouse, (whether or not the mouse button is pressed down) Windows sends a **WM_MOUSEMOVE** message. When the user releases a mouse button, Windows sends the window a **WM_LBUTTONUP** message.

In Scribble, these messages are first sent to a window, in this case the currently active view. The view uses its message map to determine whether it has a member function that can handle the message. For example, on receiving a **WM_LBUTTONDOWN** message, the view checks to see whether it has a handler associated with that message name and, if so, calls the handler.

It's appropriate that the view should handle mouse-drawing messages because it's in the view that Scribble's drawing takes place. The view represents that part of the document that can be seen at any one time.

The message handlers track mouse activity, and draw in the view accordingly. They also call member functions of the document to update its data. As the user draws a stroke, the points that make up the stroke are stored in the document's stroke list.

Connecting Messages to Code

This section takes you through the steps required to connect the three mouse-related messages needed in Scribble —**WM_LBUTTONDOWN**, **WM_LBUTTONUP**, and **WM_MOUSEMOVE**—to the equivalent message-handler member functions of class `CScribbleView`—`OnLButtonDown`, `OnLButtonUp`, and `OnMouseMove`.

This step will be different from the previous ones. Instead of directly adding lines of code to a file in the text editor, you'll first use the WizardBar to make connections between Windows messages and their handler functions. ClassWizard adds an entry to the message map in ScribbleView.cpp for class `CScribbleView`, and writes a default member function definition to the same file for the handler function. Then, as described in "Adding the Message-Handler Functions," you use the ClassView pane to jump directly to the skeleton member function, and you fill in the function's code.

If you're reading along instead of working in the files, you can still try out the WizardBar on the starter code you created with AppWizard in Chapter 3

For more information see "Using WizardBar," in Chapter 12 of the *Visual C++ User's Guide*.

▶ To connect the messages to Scribble's code

1 With ScribbleView.cpp open in the text editor, use the Messages list to select the **WM_LBUTTONDOWN** message.

Tip Type the first letters of the message to jump alphabetically through the function list. If your list-box is too narrow to display longer message names, try resizing the editor window.

The **WM_LBUTTONDOWN** message is not displayed in bold because it currently has no handler function.

2 Choose Yes when the message box appears, informing you that the message is not handled and asking if you want to add a handler.

You won't fill in the handler just yet.

3 Repeat steps 1 and 2 for the additional mouse messages: **WM_LBUTTONUP** and **WM_MOUSEMOVE**.

After you choose Yes, ClassWizard does the following things to associate each of the three messages with its handler and to greatly simplify your work:

- Adds the following function declarations for the handlers to the `CScribbleView` class declaration in file ScribbleView.h:

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

- Adds the following message-map entries for the message-to-handler connections in `CScribbleView`'s message map in file ScribbleView.cpp:

```
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
ON_WM_MOUSEMOVE()
```

- Adds the appropriate function definitions (with a default body) to file ScribbleView.cpp. For example, here is the default function definition ClassWizard added for `OnLButtonDown`:

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

  Notice that ClassWizard embeds a comment reminding you what to do and adds a call to the **OnLButtonDown** member function of class **CView,** the base class of `CScribbleView`. Also, if your files were open in the workspace window, ClassWizard marks them as modified (an asterisk appears in the title bar next to the filename of files that have not been saved). You can, if you like, save the files now, or you can let Visual C++ save them for you automatically when you build your project or exit the program.

For more information, see Chapter 2, "Working with Messages and Commands," in *Programming with MFC.*

Adding the Message-Handler Functions

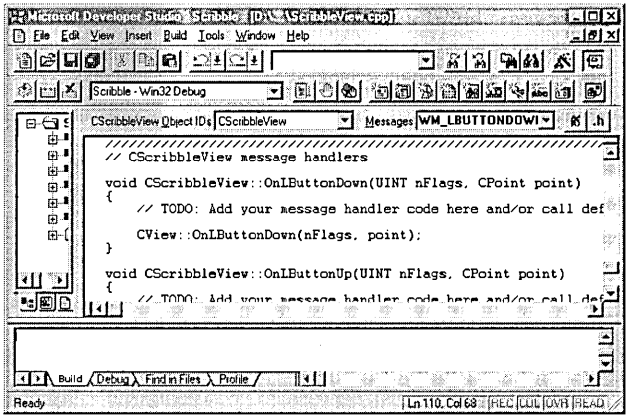
With the connections made, you can fill in the bodies of the handler functions.

▶ To fill in Scribble's message-handler function bodies

1. Use ClassView or WizardBar to jump to the skeleton member function. (For example, double-click `OnLButtonDown` from ClassView or select WM_LBUTTONDOWN from WizardBar.
2. Fill in the member function with your application-specific code.

Figure 5.2 shows `OnLButtonDown` in the editor.

Figure 5.2 The Text Editor



For Scribble, you'll fill in member functions to:

- Initiate stroke drawing
- Terminate stroke drawing
- Draw while the mouse button is down

Initiate Stroke Drawing

The `OnLButtonDown` member function, shown below, is called via the message map when Windows sends a **WM_LBUTTONDOWN** message to the view object. The function begins a new stroke, adding the current location of the mouse to the stroke and adding the stroke to the document's stroke list. Then `OnLButtonDown` captures the mouse—until the left mouse button is released to end the stroke.

▶ To add code for OnLButtonDown

1 Use ClassView to jump to the `OnLButtonDown` function definition in ScribbleView.cpp.

–or–

2 With ScribbleView.cpp open in the editor window, choose WM_LBUTTONDOWN from the WizardBar Messages list.

3 Replace the \\TODO comments and code with the code shown here:

```
// Pressing the mouse button in the view window
// starts a new stroke.

m_pStrokeCur = GetDocument( )->NewStroke( );
// Add first point to the new stroke
m_pStrokeCur->m_pointArray.Add(point);
```

```
SetCapture( );  // Capture the mouse until button up
m_ptPrev = point;  // Serves as the MoveTo( ) anchor point
                   // for the LineTo() the next point, as
                   // the user drags the mouse
return;
```

This version of `OnLButtonDown` doesn't include a call to the base class version. It completely replaces the inherited behavior.

Terminate Stroke Drawing

The `OnLButtonUp` member function, shown below, ends the current stroke when the user releases the left mouse button. The function draws a line to connect the last stroke, then releases the mouse for use by other windows. The test at the beginning calls the Windows **GetCapture** function to determine whether the current window has control of the mouse. If not, the user is not currently drawing in this view.

▶ To add code for OnLButtonUp

1 Use ClassView to jump to the `OnLButtonUp` function definition in ScribbleView.cpp.

– or –

2 With ScribbleView.cpp open in the editor window, choose WM_LBUTTONUP from the WizardBar Messages list.

3 Replace the \\TODO comments and code with the code shown here:

```
// Mouse button up is interesting in the Scribble
// application only if the user is currently drawing a new
// stroke by dragging the captured mouse.

if( GetCapture( ) != this )
    return; // If this window (view) didn't capture the
            // mouse, the user isn't drawing in this window.

CScribbleDoc* pDoc = GetDocument();
CClientDC dc( this );
CPen* pOldPen = dc.SelectObject( pDoc->GetCurrentPen( ) );
dc.MoveTo( m_ptPrev );
dc.LineTo( point );
dc.SelectObject( pOldPen );
m_pStrokeCur->m_pointArray.Add(point);

ReleaseCapture( );  // Release the mouse capture established
                    // at the beginning of the mouse drag.
return;
```

Draw While the Mouse Button Is Down

Between the time that the mouse button goes down and the time that it's released, Scribble tracks the mouse and draws a trace of its movements in the view.

OnMouseMove, shown below, is called as the user moves the mouse while drawing the current stroke. The function connects the latest mouse location with its previous location and saves the new location as the previous point for the next time the function is called. To do the drawing, OnMouseMove constructs a local **CClientDC** object used to draw in the window's client area.

▶ To add code for OnMouseMove

1 Use ClassView to jump to the OnMouseMove function definition in ScribbleView.cpp.

–or–

2 With ScribbleView.cpp open in the editor window, choose WM_MOUSEMOVE from the WizardBar Messages list.

3 Replace the \\TODO comments and code with the code shown here:

```
// Mouse movement is interesting in the Scribble application
// only if the user is currently drawing a new stroke by
// dragging the captured mouse.

if( GetCapture( ) != this )
    return;     // If this window (view) didn't capture the
                // mouse, the user isn't drawing in this window.

CClientDC dc( this );

m_pStrokeCur->m_pointArray.Add(point);

// Draw a line from the previous detected point in the mouse
// drag to the current point.
CPen* pOldPen =
        dc.SelectObject( GetDocument( )->GetCurrentPen( ) );
dc.MoveTo( m_ptPrev );
dc.LineTo( point );
dc.SelectObject( pOldPen );
m_ptPrev = point;
return;
```

In the Scribble application, OnLButtonDown, OnMouseMove, and OnLButtonUp handle the three phases of mouse drawing: beginning to track the mouse, tracking the mouse and connecting points, and ending mouse tracking.

For more information about MFC classes mentioned in this section, see the *Class Library Reference.*

Build Scribble — Step 1 Version

In this section, if you've been working along, you'll compile your completed code and try out the program. If you're simply reading along, use Books Online to preview the sample application.

Note You can easily make a local copy of the sample source code files. For more information, see "Installing the Sample Files," in Chapter 2. Note that the filenames in the sample source code may differ from the ones generated by AppWizard, depending on your specifications.

▶ To build Scribble Step 1

1 Open your Scribble project (if it's not already open) by choosing Scribble.mdp from the MRU list.

2 If necessary, specify the Debug build by doing one of the following:

- From the Default Configuration drop-down list on the Build toolbar, choose the Debug version of the project.

  –or–

- Choose Configurations from the Build menu, and then in the Configurations dialog box, select the Debug option.

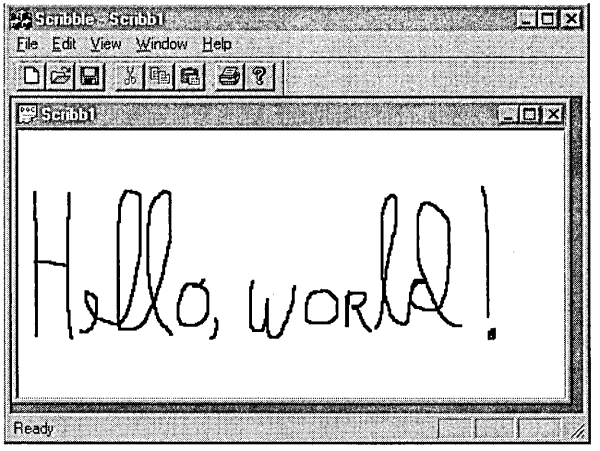
3 From the Build menu, choose Build Scribble.exe to compile and link Scribble.

▶ To give Scribble a try

1 From the Build menu, choose Execute Scribble.exe to run Scribble.

When Scribble runs, an MDI application window appears with a menu bar containing File, Edit, View, Window and Help menus and a toolbar and status bar. It has one document window open as shown in Figure 5.3.

Figure 5.3 Scribble Step 1



2 Move, resize, minimize, and maximize the document window.

3 Draw "Hello, World!" (or anything) in the window. Then save the file as HELLO.SCB.

4 Try the Print Preview and Print commands on the File menu.

Note Drawing on the printer device will show the strokes at a reduced size. This is because the current version of Scribble uses **MM_TEXT** mapping mode. Later in this tutorial, you'll use **MM_LOENGLISH** mapping mode to improve the printer output.

5 If you have an electronic mail system, try the Send command on the File menu.

6 Close HELLO.SCB and reopen it with the Open button on the toolbar, or by choosing it from the MRU list.

7 Create a new document with the New button on the toolbar and draw in the new document. (Save the new document if you like.)

8 Exit Scribble.

This concludes your quick introduction to Scribble. You've seen how to implement the document with serialization code, and the view with message handling functions. In Chapter 6, "Constructing the User Interface," you'll learn how to use the resource editors to construct some additional user-interface components. In later chapters you will learn to add more (and more interesting) code to Scribble.

CHAPTER 6

Constructing the User Interface

The next three chapters show how to use some of the powerful tools supplied with Microsoft Developer Studio and the Microsoft Foundation Class Library (MFC).

- This chapter explains how to use resource editors to customize Scribble's default menus and toolbar.
- Chapter 7, "Binding Visual Objects to Code Using WizardBar," describes how to use WizardBar and the text editor to bind Scribble menu items and toolbar buttons to commands, and define message-handler functions to process the commands.
- Chapter 8, "Adding a Dialog Box," shows how to use the dialog resource editor and WizardBar to create a dialog box and connect it to a menu command.

This chapter and Chapter 7 cover Step 2 of Scribble. You can build the new version of Scribble at the end of Chapter 7.

If you are working along, begin with the files in your Scribble project directory. At this point, your files should be very similar to the sample source files in the SCRIBBLE\STEP1 subdirectory.

Note If you have not made a local copy of the sample source code for this tutorial step and you wish to do so, see "Installing the Sample Files," in Chapter 2.

As you read this chapter, perform all steps that use the resource editors. At the end, your resource file, Scribble.rc, should closely resemble the same file in the SCRIBBLE\STEP2 sample source directory. You'll also use the text editor to make a small addition to MainFrm.cpp.

If, on the other hand, you want to read along without adding code, refer to the files in the SCRIBBLE\STEP2 subdirectory. You can also easily preview a running version of Scribble at this tutorial step. For more information, see "Previewing the Sample Applications," in Chapter 2.

Even if you don't want to add code, it's a good idea to work along in this chapter to familiarize yourself with the resource editors and the Visual C++ programming process.

Edit Scribble's Menus

The first task in this chapter is to edit Scribble's default menus by using the menu editor.

Thanks to AppWizard, Scribble starts out with a skeleton resource file, Scribble.rc, with no effort on your part. Scribble.rc already contains the default menus described in the next section, as well as other resources AppWizard generates by default, such as a toolbar, an icon to depict the application in its minimized state, and so on. You can open Scribble.rc and examine the default menus and other resources if you like.

For more information about editing resources, see "Using the Resource Editors," in Chapter 5 of the *Visual C++ User's Guide*.

▶ To examine the contents of Scribble.rc

1. From ResourceView, double-click Scribble.rc to open it.

   This launches the resource compiler. Once Scribble's resource file has been compiled, Scribble's resource types appear as folder icons in ResourceView.

2. Expand any of the folders to open them and view the resources they contain; then double-click a resource to open it inside a resource editor window, where you can both view and edit it.

Default Menus

The menu resources AppWizard generated by default for Scribble include:

- A menu bar to display when Scribble, a multiple document interface (MDI) application, has no documents open.

  The menus include a basic File menu, a View menu for toggling the visibility of Scribble's status bar and toolbar, and a basic Help menu.

- A menu bar to display when a Scribble document is open.

  They include, besides those above, more File menu commands, an Edit menu with standard commands, and a Window menu with standard commands (supplied only for MDI applications, like Scribble).

Note Single document interface (SDI) applications have only one menu bar. The correct menu bars are generated when you choose between single-document and multiple-document options in AppWizard.

Scribble's New Menu Commands

The goal in this section is to add a new Clear All command to Scribble's default Edit menu, as well as a completely new Pen menu with two commands, Thick Line and Pen Widths. The Clear All command clears the current drawing and deletes its stroke data. The Thick Line command toggles the thickness of the lines (either thin or

thick) used to draw subsequent strokes. The Pen Widths command displays a dialog box that lets the user define, in pixels, the width of the thick and and thin lines for subsequent drawing.

The next section, "Adding the Menus," describes how to create the new menu items. In Chapter 7, "Binding Visual Objects to Code Using WizardBar," you'll see how to use WizardBar to connect the menus to code. And, in Chapter 8, "Adding a Dialog Box," you'll see how to create the Pen Widths dialog box and connect it to its associated menu command.

Adding the Menus

This section describes how to add the new Scribble menu items, demonstrating some fundamental techniques for using the menu editor to edit menu resources. For more information, see Chapter 7, "Using the Menu Editor," in the *Visual C++ User's Guide*.

Add the Clear All Command to Scribble's Edit Menu

When you add the new Clear All menu item to Scribble, you'll learn how the menu editor works. If you're working along, use the following procedure.

▶ To add Scribble's Clear All menu command

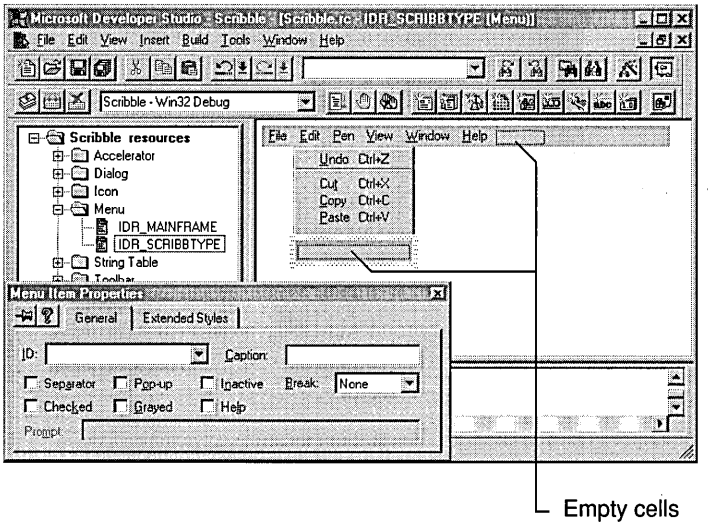
1. If you don't already have Scribble's resource file open, switch to ResourceView and expand the Scribble folder.
2. Expand the Menu folder.

   Two menu IDs appear: IDR_MAINFRAME and IDR_SCRIBBTYPE. These menus correspond to the default menus discussed previously. Their IDs are defined for you by AppWizard, based on the document type you chose (MDI or SDI).
3. Double-click IDR_SCRIBBTYPE.

   The menu editor opens, with the menu displayed much as it appears in the running application.
4. Click the Edit menu item.

   The Edit menu drops down as it would in the application. An empty cell sits below the last menu item, as shown in Figure 6.1. This cell defines where you add the next menu item.
5. Click the cell at the bottom of the Edit menu to select it.

Figure 6.1 Menu Editor for IDR_SCRIBBTYPE



6 Type the caption `Clear &All`.

When you start typing, the Properties page for menu items opens, with your cursor in the Caption edit box. As you continue to type, the caption appears both in the Caption edit box and on the Edit menu.

Note To ensure that the Properties window remains open while you are editing Scribble's menus, press the pushpin in the upper-left corner of the Properties window.

Typing the ampersand character (&) in front of a letter creates an access key combination. As you type &A, for example, the letter A appears underlined in the menu, which means that simply pressing A with the Edit menu open chooses the Clear All command. (If an access key appears in a top-level menu item, you can press ALT plus the letter to open the menu.)

Note To specify an accelerator, or shortcut key, for the menu item, append its specifier after the caption. For example, to specify CTRL+O as the accelerator for an Open command, the caption string would read "Open...\tCTRL+O" where "\t" signifies a tab to align the column.

7 Open the ID drop-down list in the Properties window and begin typing the ID for the Clear All command: `ID_EDIT_C`.

If you type with the drop-down list open, you'll see the list box scroll to the first ID that matches the letters you've typed so far. Visual C++ always ensures that the ID you enter is unique. **ID_EDIT_CLEAR_ALL** is a command ID predefined by the class library. If what you type matches a predefined command, Visual C++ assigns the predefined command ID to your menu item rather than creating a duplicate.

Several IDs that begin with "ID_EDIT_C" appear in the list box.

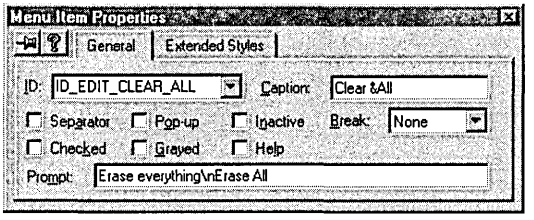
8 Select the ID_EDIT_CLEAR_ALL entry.

As soon as you select the predefined ID, the following string appears in the Prompt edit box: "Erase everything\nErase All". This prompt string is displayed in the status bar, if the application has one, when the user navigates up and down the menu using the keyboard; the text after the newline character (\n) appears as a tool tip if you create a toolbar button for this menu command. (For more information, see "Add the Thick Line Button to Scribble's Toolbar Bitmap," later in this chapter).

AppWizard predefines the prompt text for the **ID_EDIT_CLEAR_ALL** symbol. For any ID that isn't predefined, you should enter a descriptive prompt string. This context-sensitive menu information is essentially free, so take advantage of it.

Figure 6.2 shows the property page after you've selected the ID.

Figure 6.2 Property Page with ID



You can define your own command IDs, of course. You'll see an example under "Add Scribble's Pen Menu."

9 Change the Prompt wording to "Clears the drawing".

Since this menu command won't have an equivalent toolbar button, you remove the tool tip.

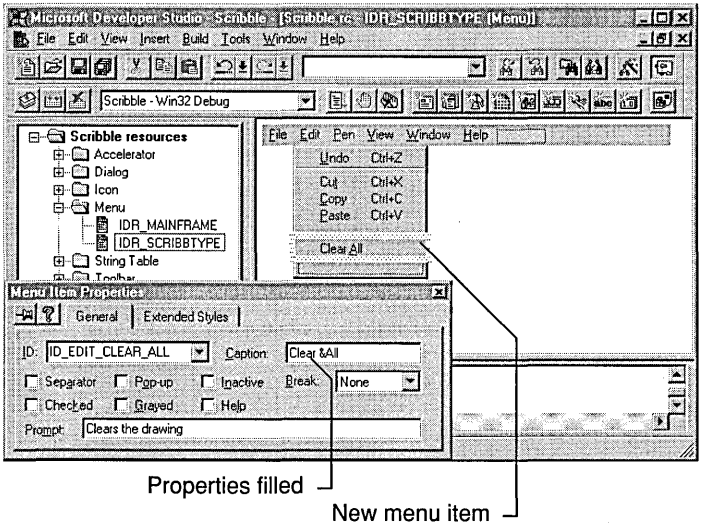
10 Save the resource file.

That's it. You've added the Clear All command to the Edit menu.

Note You don't have to press ENTER or click any buttons to accept the values you've entered in the menu Properties page. Your entries are stored immediately by the program. For more information, see "Using Property Pages" in the *Visual C++ User's Guide*.

Figure 6.3 shows how the menu looks at this stage.

Figure 6.3 The Clear All Menu Item



Menu IDs and the Framework

The most important thing about defining the menu command is assigning it an ID. To the framework, the ID *is* the command. At some point, you have to specify what happens when the user chooses the Clear All menu command; that is, which code will be executed? You'll learn more about commands in the next chapter.

Add Scribble's Pen Menu

Adding a new menu is similar to adding new commands to existing menus. If you're working along, use the following procedure.

Note This procedure assumes you have "pinned" the Menu Properties window so that it remains open.

▶ To add Scribble's Pen menu

1. With the menu-editor window still showing, click in the empty cell at the right-hand end of the Scribble menu bar (after the Help menu).
2. To position the menu entry, drag the selected cell to the left and drop it between the Edit and View menus.).

3 Type the new menu's caption: `&Pen`.

The caption appears simultaneously in the cell and in the Caption edit box of the Properties window.

4 Press ENTER to advance to the first menu item on the Pen menu (or click the empty cell that descends beneath the word "Pen").

5 Type this menu item's caption: `Thick &Line`.

Move your cursor to the ID box in the Menu Properties window. You'll notice that Visual C++ created an ID for you, based on the menu name and the menu item name: `ID_PEN_THICKLINE`.

6 Modify this ID slightly, to ID_PEN_THICK_OR_THIN.

7 In the Prompt edit box, type the following command prompt string :

```
Toggles the line thickness between thin and thick
```

No default prompt string appeared because ID_PEN_THICK_OR_THIN is not a predefined command ID.

8 Select the empty cell at the bottom of the Pen menu, below "Thick Line." (You can also toggle between the Menu Properties window and the menu editor by pressing ENTER.)

9 Type this menu item's caption: `Pen &Widths...`.

An ellipsis (. . .) included in a menu item's caption lets the user know that selecting the item opens a dialog box or a cascading menu.

10 In the ID box, modify the Visual C++-generated ID slightly, to ID_PEN_WIDTHS.

11 Type the following command prompt string:

```
Sets the size of the thin and thick pen
```

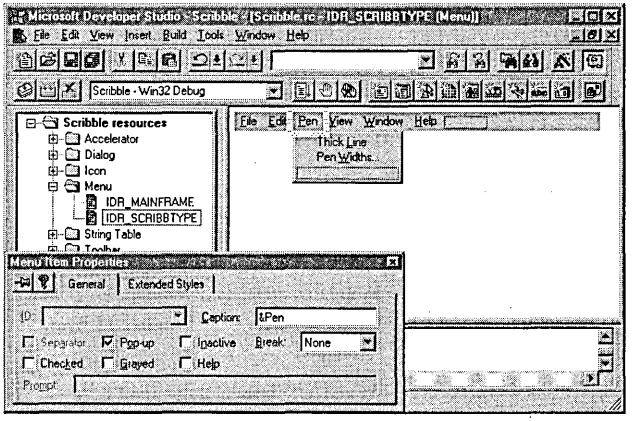
12 Close the menu editor.

If the menu editor window is maximized, you must first resize it so the control menu appears.

That's all it takes to create the Pen menu. If you like, you can save Scribble.rc before proceeding to the next step.

Figure 6.4 shows the completed menu as it appears in the menu editor.

Figure 6.4 The Completed Pen Menu



Connect the Menus to Code

Typically, at this point you would use ClassWizard or WizardBar to bind the menu commands to message-handler functions. That step is postponed until the next chapter in order to keep this chapter focused on constructing the user interface. If you like, you can skip ahead, perform the command-binding steps in Chapter 7, and then return to this chapter to edit Scribble's toolbar.

Edit Scribble's Toolbar

The resource file that AppWizard creates for Scribble, Scribble.rc, also includes a toolbar resource, shown in Figure 6.5. When you build Scribble, the framework uses the toolbar resource to create a dockable toolbar. In this section you'll use the toolbar editor to add a new button and, optionally, delete some unnecessary buttons from Scribble's toolbar.

Figure 6.5 The Default Scribble Toolbar

Earlier in the chapter, you added the Pen menu. One of its menu items is the Thick Line command. In this section, you'll add a corresponding Thick Line button to Scribble's toolbar. Then, in Chapter 7, you'll use WizardBar to connect both the Thick Line menu item and the Thick Line toolbar button to the same handler member function. Thus, the Thick Line toolbar button will become an alternative user interface for the Thick Line menu item. That is, both user-interface objects will have the same command ID so they generate the same command message, which calls the same handler function.

When the user chooses either the menu item or the toolbar button, the handler function toggles Scribble's drawing pen between thin and thick lines. Figure 6.6, at the end of this topic, shows Scribble as it appears with the finished toolbar.

About the Toolbar

Some of the buttons on Scribble's toolbar already work, as you saw in Chapter 5 when you compiled Scribble. The buttons for opening and saving files are already connected to handlers defined by the framework. All you had to do to make the file operations functional was write the serialize functions for the document and the stroke data structure. Also, the print button is supported by default, and the About button automatically displays the About box for Scribble (the About box was created for you by AppWizard).

The Cut, Copy, and Paste buttons on the toolbar appear grayed, because they have no handlers defined for them. These buttons will not be implemented for Scribble, and so you will (optionally) remove them from the Scribble toolbar.

Add the Thick Line Button to Scribble's Toolbar Bitmap

You'll add the Thick Line button to Scribble's toolbar by performing a few simple steps:

- Open Scribble's toolbar resource
- Delete toolbar buttons
- Add a new button
- Assign a resource ID to the new button
- Add a tool tip

For more information, see Chapter 9, "Using the Toolbar Editor," in the *Visual C++ User's Guide.*

Opening Scribble's Toolbar Resource

As with any of the Visual C++ resource editors, when you open a specific application resource, the corresponding resource editor opens automatically.

▶ To open Scribble's default toolbar resource

1. If you don't already have Scribble's resource file open, switch to ResourceView and expand the Scribble folder.
2. Expand the Toolbar folder and double-click IDR_MAINFRAME.

   The toolbar editor, shown in Figure 6.7, opens, displaying the default toolbar resource that AppWizard created for Scribble. The first button on the toolbar, selected by default, appears in the bottom pane, or magnified view, of the editor window.

The graphics and color tools, shown in Figure 6.6, also open as part of the toolbar editor. If these graphics tools don't appear, choose Toolbars from the View menu and select Graphics and Colors in the dialog box. You can drag the graphics tools to either side of the screen and dock them to get a better view of the editor window.)

Figure 6.6 The Graphics Toolbar

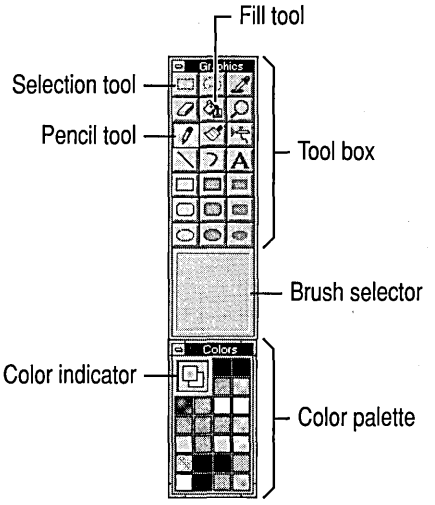
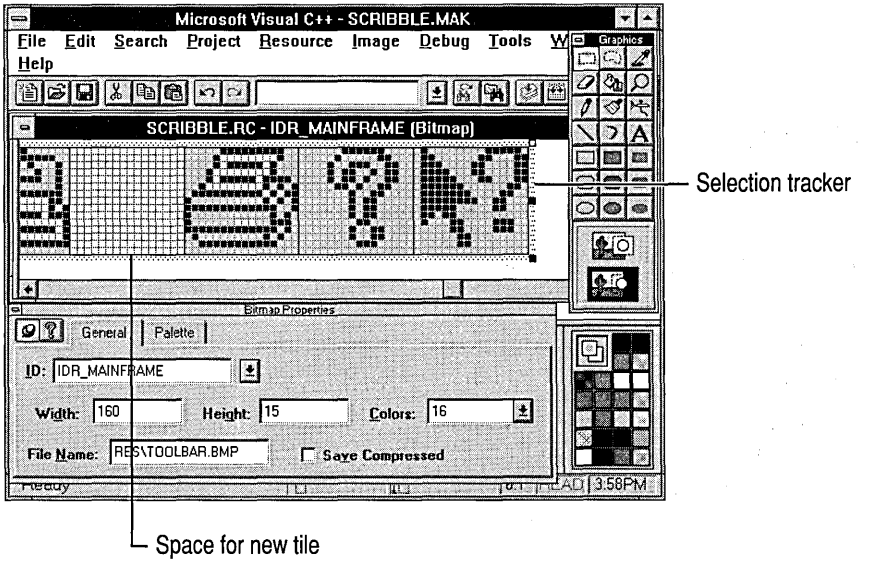


Figure 6.7 The Toolbar Editor



Deleting and Adding Buttons to Scribble's Toolbar

Adding, copying, moving, and deleting toolbar buttons are all very simple operations.

▶ To delete a toolbar button

- Drag the button off the toolbar (in the top, or normal view pane).

  In this case, drag the Cut, Copy, and Paste buttons off the Scribble toolbar.

  This step is optional; if you don't remove these buttons, they will remain grayed in the running application but otherwise will not interfere with Scribble operations.

▶ To add the new toolbar button

1. Select the blank button at the right-hand end of the toolbar resource.

   It receives focus in the two split panes of the editing window.

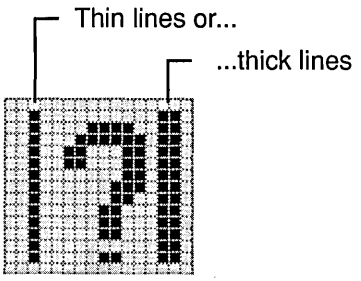
   Tip If you want the button to appear larger in the editor, choose the Magnify tool and select the magnification factor you want.

2. Choose the pencil tool from the graphics toolbar.
3. Using the magnified view of the button, draw the image shown in Figure 6.8.

   It doesn't have to be exact. If you make a mistake, use the eraser tool.

4. Save your edits.

Figure 6.8 Thick Line Toolbar Button Resource



Note The blank button that appears by default in the toolbar editor window does not appear in the running application.

Associating the Toolbar Button with a Command ID

In the next step you'll associate the new Thick Line button with a command ID so that the button works when chosen in the running Scribble application. This step is identical to the one you performed to associate a menu item with a command ID.

You'll bind the Thick Line button to `ID_PEN_THICK_OR_THIN`. You defined that ID earlier for the Thick Line menu command, so Visual C++ has already written a **#define** for the ID in the project file called Resource.h. Your only task at this point is to associate the ID with the button.

Note You don't need to delete the command IDs for the buttons you deleted; Visual C++ does that for you automatically.

▶ To associate the button with a command ID

1 If necessary, choose Properties from the Edit menu to display the Toolbar Button Properties page.

You'll notice that Visual C++ has already assigned a command ID to the button. You could accept (or modify) this ID, but in this case we want to select an existing ID — the one defined for the Thick Line menu command.

2 From the ID drop-down list, select ID_PEN_THICK_OR_THIN.

You can also type this directly into the ID edit box.

3 Save the .rc file.

By associating the command ID with the toolbar button, the string resources become active for the button as well: when the mouse passes over the button, the prompt string displays in the status line, and the tool tip displays by the button.

Figure 6.9 shows the modified toolbar resource as it appears in Scribble.

Figure 6.9 The Edited Toolbar Resource

Adding a Tool Tip

It's easy to add a tool tip, the yellow hint that appears when the mouse rests over an interface element, to your new toolbar button.

Note AppWizard appends the tool tip string automatically to the prompt strings it generates.

▶ To add a tool tip

1 If necessary, select the new toolbar button in the editor window, and choose Properties from the Edit menu to display the Toolbar Button Properties page.

In the Prompt: box, you'll see the text string "Toggles the line thickness between thin and thick". You entered this text for the Thick or Thin menu item on Scribble's Pen menu; it appears in the status line when the mouse passes over the menu command.

2 To add a tool tip, at the end of the Prompt text type a newline character (`\n`) plus the text you want displayed in the tip (there should be no space between the newline character and the tip text). If you want a tool tip without a prompt string, simply start with the newline character.

Keep this text short. For Scribble, type `\nToggle Pen` after the existing Prompt string.

Summary: Constructing the User Interface

Scribble's resource needs are simple, so this chapter introduced only a few of the things you can do with the resource editors in Visual C++. For information about their many capabilities, see "Using the Resource Editors" in Chapter 5 of the *Visual C++ User's Guide*.

After editing your application's menus and toolbar, the next step is to connect them to code using WizardBar and ClassView. That step is explained in Chapter 7.

CHAPTER 7

Binding Visual Objects to Code Using WizardBar

Like all applications written for Windows, Scribble is message driven. A keystroke, mouse click, or other event causes messages to be sent to some part of the application that can respond to the event. In Chapter 5, for example, you saw that Scribble implements mouse drawing by detecting and responding to messages generated by mouse clicks and drags.

This chapter discusses a category of messages called "commands," which are messages to your application from menu items, toolbar buttons, and accelerator keys. For more information, see Chapter 2, "Working with Messages and Commands," in *Programming with MFC.*

The expanded version of Scribble developed in this chapter adds two menu items — one that generates commands to toggle the line thickness for drawing and one that clears all strokes from the current document. The command that toggles line thickness is also duplicated by a button on Scribble's toolbar.

You created the resources for Scribble's new menu items and its new toolbar button in Chapter 6. Now you can use WizardBar to assign a user-interface object, such as a menu item, to a command and map the command to a function that handles it.

In this chapter, you will:

- Extend your knowledge of ClassWizard and WizardBar, begun in Chapter 5.
- Add new command-handling code for Scribble.
- Connect a toolbar button and a menu item to the same command.
- Learn how to keep your user-interface objects (menus and toolbar buttons) updated in response to changing program conditions by, for example, enabling or disabling a menu item and checking or unchecking a button.

  (For more information, see "How to Update User-Interface Objects" in *Programming with MFC.*)

Completing Scribble Step 2

This chapter and the previous chapter cover Step 2 of Scribble. If you are working along, begin with the files in your Scribble project directory.

After following the steps descibed in this chapter, your files should closely resemble the files in the SCRIBBLE\STEP2 subdirectory.

If, on the other hand, you want to read along without adding code, refer to the files in the SCRIBBLE\STEP2 subdirectory.

Note If you have not made a local copy of the sample source code for this tutorial step and you wish to do so, see "Installing the Sample Files" in Chapter 2.

You can also easily preview a running version of Scribble as it appears when this tutorial step is completed. For more information, see "Previewing the Sample Applications" in Chapter 2.

What ClassWizard and WizardBar Can Do

ClassWizard and WizardBar are tools you'll find yourself using quite frequently as you program in the Developer Studio environment. WizardBar provides a shortcut to many of the tasks you can perform with ClassWizard, such as:

- Connect standard Windows messages to message-handler functions.
- Connect user-interface objects to message-handler functions.
- Edit existing message maps and message-handler functions.

In this chapter, you'll learn to use WizardBar to bind commands to message-handler functions.

For more information, see the topics "Using WizardBar" and "Using ClassWizard," in Chapter 14 of the *Visual C++ User's Guide*, and the article "ClassWizard," in Part 2 of *Programming with MFC*.

Creating Message-Handler Functions with WizardBar

When you use WizardBar to create a message-handler function, ClassWizard writes an entry for the command in the chosen class's message map and adds a function declaration to the class. Also, ClassWizard writes a function template—a complete member function definition with an empty function body—in the source files that contain the class. WizardBar jumps you directly to the text editor to fill in the function template.

Important If you delete a command binding with ClassWizard, its message-map entry is deleted, but the message-handler function, and any references to it in your other code, are not deleted. You must delete those items by hand. This is for your safety; the message-handler function code, which you probably wrote, is preserved until you delete it.

For more information on command handling, see "Working with Messages and Commands," in Chapter 2 of *Programming with MFC.*

Binding Scribble's Commands

This section explains the issues and procedures involved in binding Scribble's Clear All and Thick Line commands to their handlers using WizardBar. (You'll bind the Pen Widths command in the next chapter.)

Which Command-Target Class Gets the Handler?

Before you can bind Scribble's Clear All command to a message-handler function in the document class, there are some problems to solve. Where should you put the handler for a command? Where should you put attributes, such as a line thickness value? In the document class? In the view class? Somewhere else?

Consider the specific case of Scribble. Scribble has one document class (some applications might have several kinds of documents—such as text documents and graphics documents) and one view class (some documents might have more than one way to view their data—for example, as text or as an outline).

Scribble's Clear All command has two effects: It deletes data in the document and it causes the view to be redrawn with no strokes. Should the handler for Clear All be located in the document or the view? Scribble's `CScribbleDoc` class houses the application's data structure, the stroke list. Clear All's primary effect is to delete the data. Redrawing the view afterward is secondary. Hence, it makes sense to locate the `OnEditClearAll` handler in the document.

Scribble's Thick Line command is more interesting. This command toggles the current value of a line thickness variable between thick and thin. Should the handler for Thick Line be located in the view because it affects how Scribble's data is drawn? This seems reasonable, but consider what happens when, in Chapter 9, Scribble gets splitter window functionality. In that case, each pane of the splitter window is really a new view on the same data. Should each of these views house its own line thickness information (and its own pen)? It seems a better solution is to store that information in the document instead, where all of the views can access it.

Keep in mind that this is a decision specific to Scribble's user interface, where it's desirable that the pen width commands apply to all views, not just the one with the current focus. You might choose to organize things differently in another application. This type of program flow is up to you.

Now consider a hypothetical application with more than one view on a document and perhaps even more than one frame window for the same document. Should handlers and attributes be part of the document, part of one of the frame windows, or part of one of the views? Should an attribute be duplicated in more than one view or frame window?

Here are some guidelines that may help:

- In general, put handlers in the command-target class where they have the greatest effect.
- When attributes are shared by multiple views or frame windows, put them in the common document.
- If attributes are not shared, put them in the view(s) or window(s) that use them.

Bind Scribble's Clear All Command to its Handler Code

As discussed in the previous section, "Which Command-Target Class Gets the Handler?," Scribble's Clear All command is bound to the document class.

If you're working along, use the following procedure.

▶ To create the skeleton handler for Scribble's Clear All command

1 From FileView, expand the Scribble project folder, if necessary, and double-click the icon for ScribbleDoc.cpp to open the file.

Recall the decision to handle the command from the document rather than the view. That's why the handler for Clear All will be placed in `CScribbleDoc`.

2 From the Object IDs list in WizardBar, select ID_EDIT_CLEAR_ALL.

3 From the WizardBar Messages list-box, select COMMAND.

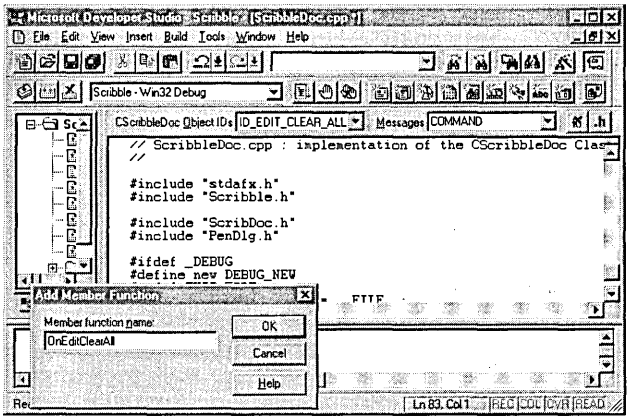
You can see both **COMMAND** and **UPDATE_COMMAND_UI** in the Messages list box. These are the two events for which the framework provides ClassWizard support in creating your command handler code. That's why, for commands, these are always the choices you see in the Messages list. In other cases, you might see other things listed—a list of Windows messages, for example, when the selected item is the name of a window or view class.

Later in the chapter, in the section "Update Scribble's Clear All Menu Item," you'll see how **UPDATE_COMMAND_UI** is used for this menu command.

If handler functions exist for these events, the functions appear in bold. Since no handler function exists for **COMMAND**, WizardBar prompts you with a message box that asks whether you want to create one.

Figure 7.1 shows the selections from steps 2, 3, and 4.

Figure 7.1 Clear All in WizardBar



4 In the Add Member Function dialog box, choose the OK button to accept the name OnEditClearAll.

ClassWizard creates the skeleton handler function in the implementation (.cpp) file, opening the file in a text editor window and highlighting the comment code so you can simply begin typing your application-specific handler code if you like. The code that you'll fill in here is described in the next procedure, "To complete the OnEditClearAll handler function."

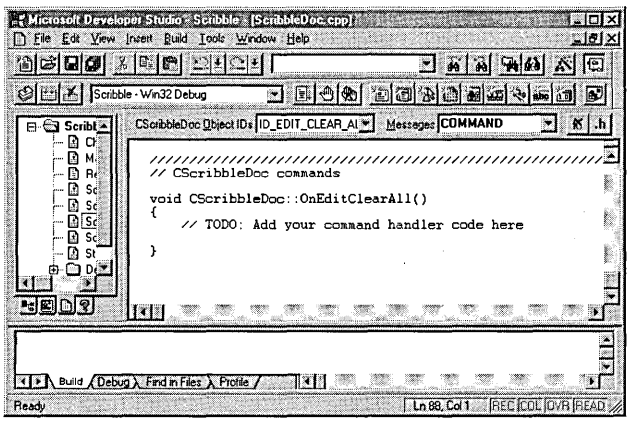
Member Function Code that ClassWizard Creates for You

ClassWizard adds several things to your source files when you use WizardBar to create a member function to handle a command:

- A message-map entry to the class's message map (in the .cpp file for the class)
- A member function declaration to the class declaration (in the .h file for the class)
- An empty member function definition to the .cpp file

Figure 7.2 shows the skeleton code for `OnEditClearAll`.

Figure 7.2 The OnEditClearAll Function Template



▶ To complete the OnEditClearAll handler function

- Add the following code to fill in the `OnEditClearAll` message-handler function. (Replace the highlighted \\TODO comments.)

```
DeleteContents( );
SetModifiedFlag();
UpdateAllViews( NULL );
```

SetModified Flag is a member function of class **CDocument**. It marks the document as changed so the framework will prompt the user to save the document when it closes.

The new `OnEditClearAll` message handler first calls `DeleteContents` to destroy the document's stroke data. (Scribble's version of `DeleteContents`, from Chapter 9, overrides **CDocument**'s **DeleteContents** member function.) Then `OnEditClearAll` calls the **UpdateAllViews** member function inherited from **CDocument** to cause all views of the data to be updated. The document's view is redrawn, this time with no data. **UpdateAllViews** takes a **NULL** argument because the document is modifying itself. The parameter is normally used to pass a pointer to the view that modified the document, but that doesn't apply here.

The `DeleteContents` member function iterates through the list of strokes. For each stroke, it gets the next stroke and calls the **delete** operator on it. For more information about working with list classes, see the article "Collections" in Part 2 of *Programming with MFC*.

Bind Scribble's Thick Line Command

When you finish adding `OnEditClearAll`, you're still in the text editor. To continue binding commands, you'll continue to choose them from WizardBar.

Like the Clear All command, the Thick Line command will be handled by the document. Recall the discussion under "Which Command-Target Class Gets the Handler?."

▶ To bind Scribble's Thick Line command

1 Make sure that file ScribbleDoc.cpp is active in the text editor.

2 In the WizardBar Object IDs list-box, select ID_PEN_THICK_OR_THIN.

3 In the Messages list box, select COMMAND.

4 In the Add Member Function dialog box, click the OK button to accept the name OnPenThickOrThin.

ClassWizard creates and displays the function template for `OnPenThickOrThin`.

5 Replace the highlighted comment text with the following code:

```
// Toggle the state of the pen between thin and thick.
m_bThickPen = !m_bThickPen;

// Change the current pen to reflect the new width.
ReplacePen( );
```

The `OnPenThickOrThin` message handler first toggles the state of a Boolean variable, `m_bThickPen`. If the variable is now **TRUE**, the pen will be thick. Otherwise, it will be thin. The handler then calls a helper function, `ReplacePen`, to reset the current pen to the new width. (You declare the `m_bThickPen` variable later in the tutorial.)

The next section describes how to create this helper function.

Adding the ReplacePen Helper Function

`ReplacePen` is not a message handler, so you don't add it with WizardBar. You could type the function directly into the text editor, by placing a definition in file ScribbleDoc.cpp and a declaration in file ScribbleDoc.h. However, ClassWizard provides a simpler way to add a new member function.

When you use the Add Function menu command, you simply enter information into a dialog box, and ClassWizard creates both the function declaration and the skeleton definition for you. All you need to do is fill in your application-specific code.

▶ To create the ReplacePen helper function

1 In ClassView, point your mouse cursor at the icon for class `CScribbleDoc`, and click the right mouse button.

Pointing to the class icon specifies the class to which your selection from the local menu applies.

2 From the pop-up menu, choose Add Function.

The Add Member Function dialog box appears.

3 In the Function Type edit box, type the function's return type (in this case, `void`).

4 In the Function Declaration edit box, type the declaration (function name and parameters, if any) of the new function.

In this case, type the following:

```
ReplacePen ()
```

There's no need to type a semi colon. Also, for functions such as this that take no parameters, the parentheses are optional.

5 In the Access area, select Protected.

6 Click OK.

Visual C++ adds the declaration to the beginning of the first Protected section of the header file it finds (creating the section if it does not exist); creates a skeleton definition in the implementation file; and jumps you to the body of the definition, so you can begin typing your application-specific code.

7 Type the following code to fill in the function definition for `ReplacePen`:

```
m_nPenWidth = m_bThickPen ? m_nThickWidth : m_nThinWidth;
// Change the current pen to reflect the new width.
m_penCur.DeleteObject( );
m_penCur.CreatePen( PS_SOLID, m_nPenWidth, RGB(0,0,0) );
```

The `ReplacePen` member function uses the C conditional operator (**?:**) to determine the pen width and return its value. Then it calls the `DeleteObject` member function of the current pen object and creates a new solid black pen with `CreatePen`, setting its width and other attributes.

Now that you've created `ReplacePen`, you want to call it from the Scribble code that initializes the pen width. This happens in the `InitDocument` member function code.

▶ To incorporate ReplacePen into Scribble

1 In ClassView, expand class `CScribbleDoc` and jump to the `InitDocument` member function.

2 In the text editor, replace the current code with the single line that calls `ReplacePen`:

```
ReplacePen();   // Initialize pen according to current width
```

Note This is the code you replace (originally added in Chapter 5):

```
m_nPenWidth = 2; // Default 2 pixel pen width
// Solid, black pen
m_penCur.CreatePen( PS_SOLID, m_nPenWidth, RGB( 0,0,0 ) );
```

This code was incorporated into the `ReplacePen` function.

3 Save your work in both ScribbleDoc.h and ScribbleDoc.cpp.

Add New Member Variables to Scribble

In addition to storing the current pen width in `m_nPenWidth`, class `CScribbleDoc` needs to keep track of whether the pen is currently thick or thin and how "thick" and "thin" are defined (in pixels). You do this by adding new data members for the thick and thin pen values. In Chapter 8, "Connecting a Class to a Dialog Box," you will add code to allow the user to define these values with a dialog box. For now, the default values will be hard coded.

▶ To add the new data members

1 Use ClassView to jump to the class definition for `CScribbleDoc`.

Locate the section labeled "`// Attributes:`".

2 Add the following marked lines after the **protected** keyword and the existing `m_nPenWidth` declaration:

```
BOOL    m_bThickPen; // Thick currently selected or not
UINT    m_nThinWidth;   // Current definition of thin
UINT    m_nThickWidth;  // Current definition of thick
```

Tip You could also use the Add Member Variable procedure from ClassView to add these three variables.

▶ To hard code the values

1 Use ClassView to jump to `InitDocument`, and add the following code just before the call to `ReplacePen`:

```
m_bThickPen = FALSE;
m_nThinWidth = 2;      // Default thin pen is 2 pixels wide
m_nThickWidth = 5;     // Default thick pen is 5 pixels wide
```

The added code specifies that the pen is initially thin and defines the meanings of "thin" and "thick."

2 Save files ScribbleDoc.h and ScribbleDoc.cpp.

ClassView displays iconic representation of the new member functions.

Updating User-Interface Objects

You'll often change the condition of menu items in your running application to provide the user with information about current program conditions. For example, you might want certain menu items to appear dimmed (grayed) to show they're unavailable. You might want other menu items to have a check mark toggle. The same general idea holds true for toolbar buttons: you can specify that they appear enabled, disabled, or perhaps pushed in, as conditions in the program change. The framework provides a direct, command-based way to set the state of the menus and toolbar buttons. For an explanation of how this works, see Chapter 2, "Working with Messages and Commands," in *Programming with MFC*.

The next section describes how you'll update Scribble's Edit menu at runtime.

Update Scribble's Clear All Menu Item

This section presents the steps you will take to update the Clear All menu item on Scribble's Edit menu. The update command is handled by the document object, which has the necessary information on whether there are any strokes in the current drawing to clear.

Recall how you used WizardBar to create a skeleton handler for the Edit menu's Clear All command, in the section "Bind Scribble's Clear All Command;" then you filled in the handler code. Similarly, in this section you'll use WizardBar to create a skeleton handler for the update command, which you'll then fill in with Scribble-specific code.

If you're working along, perform the following procedure.

▶ To add an update handler for Scribble's Clear All menu

1. Open ScribbleDoc.cpp in an editor window.
2. In the WizardBar Object IDs list-box, select ID_EDIT_CLEAR_ALL.
3. In the Messages list-box, select UPDATE_COMMAND_UI.

   The Add Member Function dialog appears, displaying a suggested name for the handler.
4. Click the OK button to accept the name `OnUpdateEditClearAll`.

   WizardBar creates and displays the function template for `OnUpdateEditClearAll`, highlighting the comment.
5. Replace the highlighted comment text with the following code:

```
// Enable the user-interface object (menu item or tool-
// bar button) if the document is non-empty, i.e., has
// at least one stroke.
pCmdUI->Enable( !m_strokeList.IsEmpty( ) );
```

6 Save your changes to ScribbleDoc.cpp.

Notice that the **UPDATE_COMMAND_UI** entry in the Messages list is now displayed in bold, indicating that a handler exists for it.

The `OnUpdateEditClearAll` handler takes one argument, a pointer to a **CCmdUI** object that contains information about the Clear All menu item on Scribble's Edit menu.

The pointer to a **CCmdUI** object, `pCmdUI`, is used to access a **CCmdUI** member function, **Enable**. **Enable** takes one Boolean argument. In this code, the expression `!m_strokeList.IsEmpty( )` evaluates to nonzero if the document has at least one stroke to clear. If the expression evaluates to zero (no strokes), the menu item is disabled (and dimmed or grayed).

Note When the user pulls down a menu, the update handlers for all items on the menu are called before the user sees the menu displayed. Thus it's important not to perform a lot of processing in your update handlers.

When you added the update command handler for the Clear All menu item, ClassWizard wrote the following message-map entry in the document's message map in ScribbleDoc.cpp:

```
ON_UPDATE_COMMAND_UI( ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll )
```

The **ON_UPDATE_COMMAND_UI** macro resembles the **ON_COMMAND** macro for the `OnEditClearAll` message handler.

In addition, ClassWizard added a new member function declaration for `OnUpdateEditClearAll` to the `CScribbleDoc` class declaration in ScribbleDoc.h. The function declaration looks like this:

```
afx_msg void OnUpdateEditClearAll( CCmdUI* pCmdUI );
```

Update Scribble's Thick Line Menu Item

Updating the Thick Line menu is very similar to updating the Clear All menu. In this case, however, rather than enabling or disabling the menu item, the handler puts a check mark beside the item or removes an existing check mark.

If you're working along, perform the following procedure. (This procedure assumes you still have ScribbleDoc.cpp open in the editor window.)

▶ To add an update handler for the Thick Line menu

1 In the WizardBar Object IDs list box, select ID_PEN_THICK_OR_THIN.

2 In the Messages list box, select UPDATE_COMMAND_UI.

3 Choose the OK button in the Add Member Function dialog box to accept the name `OnUpdatePenThickOrThin`, and to create the handler.

4 Fill in the skeleton `OnUpdatePenThickOrThin` update handler function with the following code:

```
// Add check mark to Pen Thick Line menu item if the current
// pen width is "thick."
pCmdUI->SetCheck( m_bThickPen );
```

5 Save changes to ScribbleDoc.cpp and ScribbleDoc.h.

Rather than enabling or disabling the menu command, this handler uses the pointer to a **CCmdUI** object to call the **SetCheck** member function. **SetCheck** puts a check mark in front of the menu item's text, "Thick Line," if its argument evaluates to **TRUE**, or unchecks the menu item if **FALSE**. In this case, the expression `m_bThickPen` is a member variable of `CScribbleDoc`. It evaluates **TRUE** if the line thickness is currently set to thick. Since the value of `m_bThickPen` is passed to **SetCheck**, the effect is to toggle the menu item's check mark on or off as the line thickness changes.

The **ON_UPDATE_COMMAND_UI** message-map entry and the `OnUpdatePenThickOrThin` message handler serve to update the state of the Thick Line button on the toolbar as well as the Thick Line menu item. The code line

```
pCmdUI->SetCheck( m_bThickPen );
```

adjusts the state of the toolbar button as well as updates the checked state of the menu item. For a toolbar button, "checked" means depressed.

In this example, the user would previously have reset the line thickness. The next time the user chooses the Pen menu (or the toolbar button), the user-interface update mechanism takes care of updating the check mark to match the current thickness. Similarly, the toolbar button's state toggles between a "pressed down" appearance and a normal appearance.

As with the update handler for Clear All, ClassWizard adds a message-map entry for `OnUpdatePenThickOrThin` to the document's message map in ScribbleDoc.cpp:

```
BEGIN_MESSAGE_MAP( CScribbleDoc, CDocument )
   //{{AFX_MSG_MAP(CScribbleDoc)
   // Other message-map entries
   ON_UPDATE_COMMAND_UI(ID_PEN_THICK_OR_THIN,OnUpdatePenThickOrThin)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

ClassWizard also adds a member function declaration to the document class definition in ScribbleDoc.h:

```
afx_msg void OnUpdatePenThickOrThin( CCmdUI* pCmdUI );
```

Build Scribble — Step 2 Version

How does Scribble behave with these new commands in place? Build, and then run the new Step 2 version of Scribble to find out.

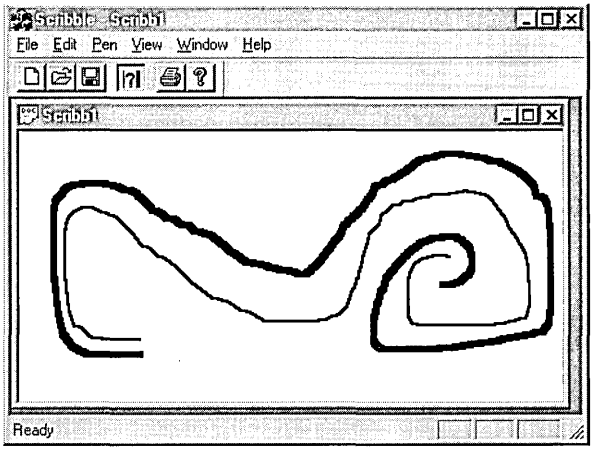
1 From the Build menu, choose Execute Scribble.exe.

2 When prompted with the message that the file Scribble.exe does not exist, choose Yes to build the file.

Once the file is built, the executable will run.

Figure 7.3 shows this version of Scribble.

Figure 7.3 Scribble Step 2



Draw some strokes with the default thin pen. Then change the line thickness by choosing the Thick Line toolbar button you added in Chapter 6, "Constructing the User Interface," and draw some new strokes. Try the same task by using the Thick Line command on the Pen menu. (The Pen Widths command is grayed because you have not hooked it up to the dialog yet. You'll do that in Chapter 8, "Adding a Dialog Box.") Clear all strokes from the drawing with the Clear All command on the Edit menu. Move the toolbar around and see how it docks when you drag it over to the frame. The framework provides this functionality for you.

Exit Scribble.

This completes Step 2 in the tutorial. You should now have a basic understanding of commands. In later chapters you'll build on that foundation.

In the next chapter, "Adding a Dialog Box," you'll implement a command that displays a dialog box and then processes the results in its message handler.

CHAPTER 8

Adding a Dialog Box

In Chapters 6 and 7, you added new commands to Scribble in two steps: first, by using the menu editor to add new menu items; and second, by using the WizardBar to define message handlers and bind them to the menu commands.

Recall that in Chapter 6, you added menu items for three new commands: Edit Clear All, Thick Pen, and Pen Widths. Chapter 7 discussed binding only the first two of these commands.

The Pen Widths command is somewhat different from the other two commands. Both the Edit Clear All and Thick Pen commands execute to completion as soon as the user selects them. By contrast, the Pen Widths command requires more information from the user. This command opens a dialog box, one that lets the user specify the widths of the Thin Pen and the Thick Pen.

Before you can write a message handler for the Pen Widths command, you have to design the dialog box that it displays and define a new class to manage the dialog box. That's what you'll do in this chapter.

This chapter develops a modal dialog box using the same general procedure that was used for adding menu commands in Chapters 6 and 7. You'll use the dialog editor to design the dialog box's appearance, and then use ClassWizard to declare a dialog class, and WizardBar to define message handlers and bind them to the dialog box.

This chapter describes the following topics:

- Designing a dialog box
- Connecting a class to a dialog box
- Opening the dialog box from your application

For more information about editing dialog boxes, see Chapter 6, "Using the Dialog Editor," in the *Visual C++ User's Guide*.

Completing Scribble Step 3

This chapter covers Step 3 of Scribble. If you want to work along, adding the code as you go, begin with the files from your Scribble project directory. At the end of the chapter, your files should closely resemble the files in the SCRIBBLE\STEP3 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or view the files in the SCRIBBLE\STEP3 subdirectory.

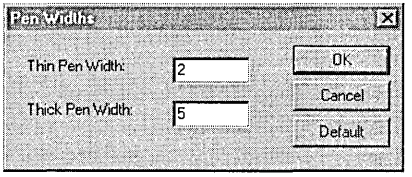
Note If you have not made a local copy of the sample source code for this tutorial step and you wish to do so, see "Installing the Sample Files" in Chapter 2.

You can also easily preview a running version of Scribble as it appears at the completion of this tutorial step. For more information, see "Previewing the Sample Applications" in Chapter 2.

Designing a Dialog Box

Figure 8.1 shows the Pen Widths dialog box that you will create.

Figure 8.1 Scribble's Pen Widths Dialog Box



The Pen Width dialog box will have the following characteristics:

- Thin and Thick Pen width edit boxes where the user can enter any number (representing pixels) between 1 and 20. If the user enters a value outside this range, Scribble displays a message box stating the legal range; after dismissing the message box, the user can enter new values.
- A Default button that enables the user to reset the pen widths to their default values.
- An OK button that enables the user to use the specified values for any subsequent drawing.
- A Cancel button that enables the user to cancel any values entered in the dialog.

Developer Studio Dialog Resource Editor

Microsoft Developer Studio provides a dialog resource editor for designing dialog boxes. This editor displays the dialog control toolbar, which shows the available controls (such as radio buttons, check boxes, and pushbuttons). You select controls from the toolbar and position them on your dialog box. You can move and resize the controls directly by using the mouse.

You use the property page for each control to specify its caption and ID.

Designing a dialog box requires three steps:

1. Creating a new dialog box and editing its caption and ID.
2. Adding the controls and editing their captions and IDs.
3. Arranging and testing the controls within the dialog box.

Create the Dialog Box

Microsoft Developer Studio provides many predesigned resources that you can easily incorporate into your projects. The default dialog box is one such resource. In this procedure, you'll create a simple dialog box by starting with the default dialog box that Developer Studio provides.

▶ To create the Pen Widths dialog box

1 With your Scribble project open, from the Insert menu, choose Resource.

2 In the Insert Resource dialog box, select Dialog from the list of resource types and choose OK.

Scribble's resource file, Scribble.rc, opens and the dialog editor window appears, displaying a default dialog box that contains two pushbuttons labeled OK and Cancel. The dialog controls toolbar also appears.

3 If the property page is not currently displayed, double-click the dialog box and then choose the pushpin button on the property page to keep it open.

4 In the ID box, type `IDD_PEN_WIDTHS`.

This is not a predefined ID, so you can't select it from the drop-down list.

5 In the caption box, change the caption to `Pen Widths`.

Notice that the title bar of the dialog box reflects the new caption.

6 Optionally, save Scribble.rc.

Note The OK and Cancel buttons have predefined properties, including their command IDs—**IDOK** and **IDCANCEL**, respectively. If you select the OK button and switch to the Styles tab of the Properties dialog, you'll see that the Default Button check box is also checked.

In the next section you'll add several controls to the default dialog box resource.

Add the Controls

This procedure assumes you have the Properties Page "pinned down," and the `IDD_PEN_WIDTHS` dialog resource open in the dialog editor.

▶ To add controls to the Pen Widths dialog box

1 From the control toolbar, add two edit box controls to the Pen Width dialog box.

- Select the first edit box to display its property page. Change its ID to `IDC_THIN_PEN_WIDTH`.
- Select the second edit box, and in the property page change its ID to `IDC_THICK_PEN_WIDTH`.

2 From the control toolbar, add two static text controls to contain the descriptions for the two edit controls.

- Select the first text box to display its property page. Change the caption to "Thin Pen Width:".
- Select the second text box, and in the property page change its caption to "Thick Pen Width:".

Notice that the text boxes automatically resize to display the text you type.

For the purposes of Scribble, you won't have to refer programmatically to the IDs of the text boxes, so you can leave them with their default values (both have the value **IDC_STATIC**).

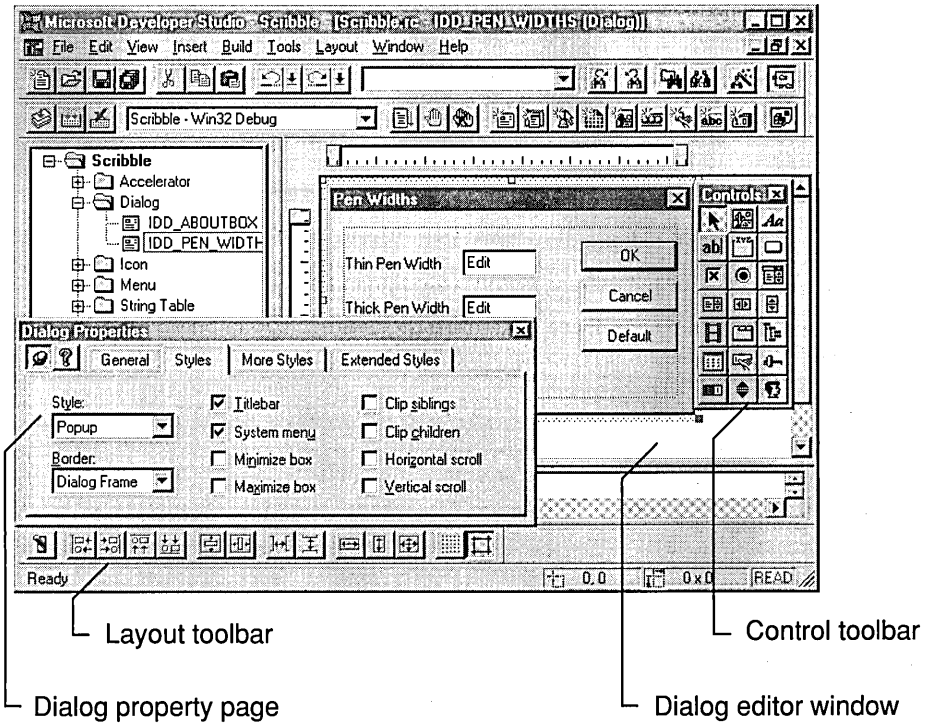
3 From the control toolbar, add a third pushbutton to the two already present.

4 Select the third pushbutton to display its property page. Change its caption to "Default" and its ID to IDC_DEFAULT_PEN_WIDTHS.

The handler for this button will reset the thick and thin pens to their default widths.

Figure 8.2 illustrates designing the Pen Widths dialog box. In this illustration, the central window is the dialog editor window. Below the dialog editor is the property page, and the control toolbar is to the right.

Figure 8.2 Designing the Pen Widths Dialog Box



Arrange and Test Controls

Once you've added all the controls to the dialog box, you can also:

- Resize the dialog box for a balanced layout.
- Align the controls, make them the same size, etc., using the commands on the Layout menu.
- Define the tab order for the controls.

  Tab order is the order in which the TAB key moves the input focus from one control to the next. You can see the tab order by choosing Tab Order from the Layout menu. To change the tab order, click each control in the order you want as the tab order.

- Test the dialog box.

  If you want to see how the dialog box will look when it's displayed, choose the Test command from the Layout menu. This displays the dialog box as it will appear in Scribble, enabling you to test aspects such as the tab order, default button, and so on. Exit Test mode by choosing either the OK or Cancel button on the dialog box or by pressing the ESC key.

Connecting a Class to a Dialog Box

Once you've specified the appearance of your dialog box, you need to specify its runtime behavior. This requires deriving a class from **CDialog** that implements your dialog box, and connecting that class to the dialog resource you created in the previous section.

In general, to connect a class to a dialog box:

1. Declare a class to represent the dialog box.
2. Declare handler functions for the messages you want to handle.
3. Map the dialog box controls to member variables of the dialog class and define what (if any) validation rules should be applied to each.

You could do all of this manually, but ClassWizard and WizardBar provide a graphical user interface that lets you do it quickly and easily. ClassWizard generates a header (.h) and an implementation (.cpp) file for your dialog class, complete with function prototypes, skeletal function definitions, a message map, and a data map.

The following sections show how these steps are accomplished for Scribble's Pen Widths dialog box.

Declare the Dialog Class

The following procedure shows you how to use ClassWizard to declare a class for the dialog box you just created. ClassWizard enables you to easily declare new classes in your application. For more information, see "Adding a Class."

Note The following procedure assumes you have the Pen Widths dialog resource open, and that you have not previously declared a class for it.

▶ To declare the new dialog class

1 From the View menu, choose ClassWizard.

–or–

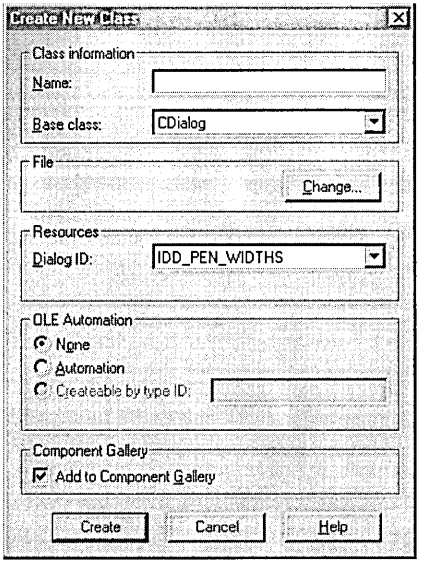
From the standard toolbar, click the ClassWizard button.

The Adding a Class dialog appears, with a message that `IDD_PEN_WIDTHS` is a new resource, and with the "Create a new class" option selected by default.

ClassWizard knows that a class hasn't been defined yet for your dialog resource, so it displays this dialog box to enable you to define one.

Note If you had created the dialog class before creating the dialog resource, you could specify the "Select an existing class option" in this dialog to connect the dialog to the existing class.

Figure 8.3 The Create New Class Dialog Box



2 Choose OK to create the dialog class.

3 The Create New Class dialog appears, with `IDD_PEN_WIDTHS` displayed in the dialog ID drop-down list under Resources.

Notice that the Base Class is already set to **CDialog**. ClassWizard assumes this is the type to use because you were using the dialog editor.

4 In the Name box under Class Information, type `CPenWidthsDlg`.

Notice, in the File box, that ClassWizard derives the name for the implementation (.cpp) and header (.h) files from the characters you type. If you wanted to change either filename you would choose the Change button.

5 Clear the Add to Component Gallery checkbox.

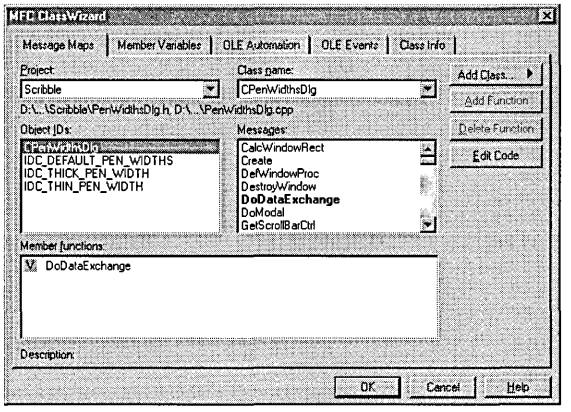
For more information about this option, see "Using the Component Gallery" in Chapter 15 of the *Visual C++ User's Guide*.

6 Click the Create button.

This creates the class and closes the Create New Class dialog box. The information for `CPenWidthsDlg` is now displayed in the ClassWizard dialog box. The Class Name edit box and Object Ids list box display "CPenWidthsDlg," and the Messages list box displays messages appropriate for the controls in the Pen Widths dialog box, as shown in Figure 8.4.

7 Click OK to close ClassWizard.

Figure 8.4 The Message Maps Tab Displaying the CPenWidthsDlg Class



Viewing the Newly Created Class

When you use ClassWizard to create a new class, ClassWizard creates and adds the associated .h and .cpp files to the project. You can view them in the FileView pane of the Project Workspace window; and, in the ClassView pane, you can view the iconic representation of the new `CPenWidthsDlg` class and its default member functions.

In the section "Declare a Message Handling Function for a Dialog Box Control," you'll use WizardBar to add a message handling function to the PenWidthsDlg.h and PenWidthsDlg.cpp files. But first, let's examine the files as they appear when first created by ClassWizard.

Header File

Here's the initial version of PenWidthsDlg.h that ClassWizard creates:

```
// PenWidthsDlg.h : header file
//

/////////////////////////
// CPenWidthsDlg dialog

class CPenWidthsDlg : public CDialog
{
// Construction
public:
    CPenWidthsDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CPenWidthsDlg)
    enum { IDD = IDD_PEN_WIDTHS };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA
```

```
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CPenWidthsDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CPenWidthsDlg)
        // NOTE: the ClassWizard will add member functions here
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

This file contains a declaration for `CPenWidthsDlg`, the class that implements the Pen Widths dialog box. At this point, the class contains two member functions: a constructor and the `DoDataExchange` function, which is described later on.

The file contains comment lines that begin `//{{AFX_` and `//}}AFX_`. ClassWizard uses those comment lines to find the sections of code that it maintains. There are three such sections in the header file, each delimited by slightly different comments: the `AFX_DATA` section, containing the declarations of the dialog data members; the `AFX_VIRTUAL` section containing declarations of override functions; and the `AFX_MSG` section, containing the declarations of the message handlers.

Note In general, you shouldn't manually edit any declarations that appear in these sections, or add code here in general. It is good style, and safe practice, to put any custom declarations in the appropriate group, but below the `//}}AFX_` line.

Implementation File

Here's the initial version of PenWidthsDlg.cpp that ClassWizard creates:

```
// PenWidthsDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Scribble.h"
#include "PenWidthsDlg.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////
// CPenWidthsDlg dialog
```

```
CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CPenWidthsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CPenWidthsDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}


void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPenWidthsDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
    //{{AFX_MSG_MAP(CPenWidthsDlg)
        // NOTE: the ClassWizard will add message map macros here
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CPenWidthsDlg message handlers
```

This file contains an empty message map and empty function definitions for the constructor and the `DoDataExchange` member function. For more information on the `DoDataExchange` function, see "Map the Controls to Member Variables," later in this chapter.

Notice that the constructor has a base initializer for **CDialog**. The **CDialog** constructor that it invokes creates a modal dialog box, and it takes two parameters: the ID of the dialog resource and a pointer to the parent window. For the first parameter ClassWizard has specified `CPenWidthsDlg::IDD`. This is an enumerated value that is defined in the `AFX_DATA` section in the class declaration. This enumerated value is equal to `IDD_PEN_WIDTHS`, the ID you specified in the section "Add the Controls." Thus the dialog class is associated with the dialog resource you created.

Also notice that the implementation file, like the header file, contains sections delimited by `//{{AFX_` and `//}}AFX_`, into which ClassWizard will insert code later.

Declare a Message-Handling Function for a Dialog Box Control

The **CDialog** class, from which `CPenWidthsDlg` is derived, defines default handlers for the OK and Cancel buttons. The Pen Widths dialog box contains a third pushbutton, the Default button. For `CPenWidthsDlg` to respond when the user chooses this button, you must define a new message handler and bind it to the Default pushbutton.

Binding a message handler to a control in a dialog box is similar to binding a message handler to a menu command, which was described in Chapter 7 in "Bind Scribble's Clear All Command to Its Handler"; both can be accomplished by using WizardBar to add an entry to a class's message map.

You should be familiar, from Chapter 7, with what the WizardBar displays for a document or view class. For a dialog class, note the following differences:

- The Object IDs box displays the IDs of all the controls in the dialog box, not the commands in a menu.
- The message being handled is a Windows control notification message, not an application-specific command. As a result, the Messages list box displays more than just **COMMAND** and **UPDATE_COMMAND_UI**; it displays all the messages that can be sent by the object that's highlighted in the Object IDs box. For example, if `IDC_THIN_PEN_WIDTH` — which is the ID of the first edit box — is highlighted in the Object IDs box, the Message box displays all the control notification messages that an edit box can generate, such as **EN_SETFOCUS**, **EN_KILLFOCUS**, and **EN_UPDATE**.

Despite these differences, the procedure for adding a message handler is the same.

Adding the Message Handler for the Default Button

The following procedure assumes you have PenWidthsDlg.cpp open in the text editor.

▶ To add a message handler for the Default button

1 In the WizardBar Object IDs list box, select `IDC_DEFAULT_PEN_WIDTHS`. This is the ID of the Default button you created.

The Messages list box now shows all the notification messages that a pushbutton can send; namely, **BN_CLICKED** and **BN_DOUBLECLICKED**.

2 In the Messages list box, select the BN_CLICKED message.

The Add Member Function dialog box appears, displaying the candidate name, "OnDefaultPenWidths." ClassWizard has synthesized this name from the object's ID and the message name.

3 Click OK to accept the function name offered by ClassWizard, and to create the function.

ClassWizard generates a skeleton function definition in PenWidthsDlg.cpp for the `OnDefaultPenWidths` message handler. Class Wizard also inserts the member function declaration into PenWidthsDlg.h, and an entry in the message map in PenWidthsDlg.cpp indicating that the member function `OnDefaultPenWidths` is the message handler called whenever the control `IDC_DEFAULT_PEN_WIDTHS` sends a **BN_CLICKED** message.

Right now the `CPenWidthsDlg` class doesn't have any member variables defined; you will define those members in the next section, "Map the Controls to Member Variables." You will implement `OnDefaultPenWidths` later, in the section "Implementing the Message Handler," after you've added the member variables.

Map the Controls to Member Variables

Scribble must be able to retrieve the values that the user enters in the Thin Pen and Thick Pen edit boxes. MFC defines a mechanism that automates the process of gathering values from a dialog box; this mechanism is called a "data map." In the same way that a message map binds a user-interface element with a member function, a data map binds a dialog-box control with a member variable. The value of the member variable reflects the status or the contents of the control. By adding entries to `CPenWidthsDlg`'s data map, you can retrieve the values entered in the Thin Pen and Thick Pen edit boxes.

For Scribble, the widths of the thin and thick pens must be between 1 and 20. You can enforce these conditions by using the automated data validation that data maps provide. If the user enters values that fall outside this range, the application displays a message box stating the legal range and allows the user to enter new values.

▶ To map the controls of the Pen Widths dialog box to member variables

1. From the View menu, choose ClassWizard, and choose the Member Variables tab.

   This tab, shown in Figure 8.5, contains a list box displaying the mapping between controls and member variables.

2. In the Class Name drop-list, select CPenWidthsDlg.

   At the moment the box displays only the IDs for the controls because you haven't yet specified which member variables the controls correspond to.

3. Select IDC_THIN_PEN_WIDTH and then choose the Add Variable button.

   The Add Member Variable dialog box appears.

4. In the Member Variable Name box, specify `m_nThinWidth` as the variable name.

5. From the Variable Type list box, choose int.

6. Choose OK to add the member variable to the class.

   Notice the changes in the dialog:

   - The member name and type you specified now appear in the Control IDs box.

- A description string of "int with validation" appears.
- Two new edit boxes (Minimum Value and Maximum Value) appear to receive the validation parameters appropriate for an integer. These correspond to the edit boxes you added to the dialog box resource.

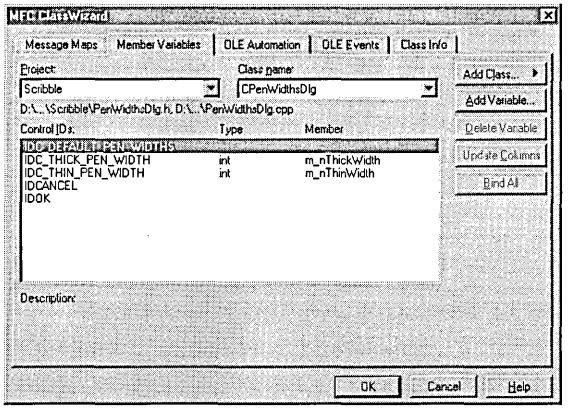
7 In the Minimum and Maximum boxes, enter 1 and `20`, respectively.

8 Repeat steps 2 through 6 for the control `IDC_THICK_PEN_WIDTH`. Specify m_nThickWidth as the member name, choose int, and enter lower and upper limits of 1 and `20`.

9 Click OK.

You've now completed the data map connecting the Pen Widths dialog box to the `CPenWidthsDlg` class. You can view the `m_ nThickWidth` and `m_nThinWidth` member variables in ClassView under the `CPenWidthsDlg` class.

Figure 8.5 The Member Variables Tab



ClassWizard inserts declarations into the data map of PenWidthsDlg.h for the member variables you specified in the Add Member Variable dialog box.

ClassWizard also makes the following changes to PenWidthsDlg.cpp after you've mapped the controls to member variables:

```
CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CPenWidthsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CPenWidthsDlg)
    m_nThinWidth = 0;
    m_ThickWidth = 0;
    //}}AFX_DATA_INIT
}
```

```
void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPenWidthsDlg)
    DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);
    DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);
    DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);
    DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
    //}}AFX_DATA_MAP
}
```

Notice that ClassWizard has initialized the member variables in the constructor and provided an implementation for the **DoDataExchange** function. The framework calls **DoDataExchange** whenever values have to be moved between the member variables in the class and the controls in the dialog box on screen (for example, when first displaying the dialog box on the screen or when the user closes the dialog box by choosing OK).

The **DoDataExchange** function is implemented using **DDX** and **DDV** function calls. A **DDX** (for Dialog Data eXchange) function specifies which control in the dialog box corresponds to a particular member variable and transfers the data between the two. A **DDV** (for Dialog Data Validation) function specifies the validation parameters for a particular member variable, ensuring that its value is legal. The **DDX** and **DDV** function calls shown above reflect the mapping and validation parameters you specified with ClassWizard.

Notice that the **DDV** function call for a given member variable immediately follows the **DDX** function call for that variable. This is a rule you must follow if you choose to manually edit the contents of the data map.

For more information about ClassWizard, see "Using ClassWizard," in Chapter 4 of the *Visual C++ User's Guide*.

Implementing the Message Handler

Recall that ClassWizard provided an empty function definition for the `OnDefaultPenWidths` message handler, which is called when the user chooses the Default button. Now that the `CPenWidthsDlg` class contains the necessary member variables, `m_nThickWidth` and `m_nThinWidth`, it's time to fill in that function definition.

The `OnDefaultPenWidths` function sets the contents of the edit boxes to the default widths of the thin and thick pens.

▶ To implement the message handler for the Default button

1 Use ClassView or WizardBar to jump to the `OnDefaultPenWidths` function definition in file PenWidthsDlg.cpp, and add the following code (you can delete the TODO comments):

```
m_nThinWidth = 2;
m_nThickWidth = 5;
UpdateData(FALSE);  // causes DoDataExchange()
    // bSave=FALSE means don't save from screen, rather, write
    // to screen
```

2 Save your changes to PenWidthsDlg.cpp.

`OnDefaultPenWidths` sets `m_nThinWidth` and `m_nThickWidth` to their default values and then calls **UpdateData**, a member function defined by **CWnd** (the base class of **CDialog**).

The **UpdateData** member function calls the `DoDataExchange` function to move values between the member variables and the controls displayed on the screen. The direction in which the data values are moved is specified by the argument to **UpdateData**. The default value of this argument is **TRUE**, which moves data from the controls to the member variables. A value of **FALSE** moves data from the member variables to the controls. The `OnDefaultPenWidths` member function passes **FALSE**, causing the default values to be displayed in the edit boxes on the screen.

Open the Dialog Box

By now you've specified almost everything about the Pen Widths dialog box: its appearance, the data map for its edit controls, and the message handlers for its pushbuttons. There's only one thing that remains to be specified: when the dialog box should be opened.

Currently there is no programmatic connection between the Pen Widths menu item and the Pen Widths dialog box. That is, the menu item and the dialog box are not bound together. You must explicitly bind them by calling the Pen Widths dialog box from within the message handler for the Pen Widths command.

How do you open a dialog box? The first step is to declare a `CPenWidthsDlg` object. This doesn't display the dialog box on the screen; it just constructs the C++ object that manages the dialog box. The second step is to complete the `OnPenWidths` member function handler for the Pen Widths menu command.

To specify that the Pen Widths command displays the dialog box modally, you call the **DoModal** member function defined by the **CDialog** class. (To display a modeless dialog, you would call the **Create** member function of **CDialog**.)

The **DoModal** function continues executing as long as the dialog box is displayed on the screen. When the user chooses the OK or Cancel button, the **DoModal** function returns **IDOK** or **IDCANCEL**, respectively, and the application can continue.

Before you write the message handler for the Pen Widths command, you need to decide which class should get the handler. Recall that in Chapter 7, in "Add New Member Variables to Scribble," you added declarations for the `m_nThickWidth` and `m_nThinWidth` member variables to the `CScribbDoc` class, because the document needs to keep track of the widths of the thick and thin pens (this allows multiple views to share the same pen widths). Since the document class has to maintain those values, it should get the handler for the Pen Widths command.

Declaring the CPenWidthsDlg Object

In the following procedure you'll use the WizardBar to add a function handler for the `OnPenWidths` message and bind the function to its handler code, which is executed whenever the user chooses the Pen Widths command.

▶ To declare the CPenWidthsDlg object

1 Open ScribbleDoc.cpp in the text editor.

2 In the WizardBar Object IDs list box, select the ID_PEN_WIDTHS command.

3 In the Messages list box, select COMMAND.

4 In the Add Member Function dialog box, choose OK to accept the candidate name "OnPenWidths."

5 In place of the highlighted \\TODO comment, add the following code:

```
CPenWidthsDlg dlg;
// Initialize dialog data
dlg.m_nThinWidth = m_nThinWidth;
dlg.m_nThickWidth = m_nThickWidth;

// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    // retrieve the dialog data
    m_nThinWidth = dlg.m_nThinWidth;
    m_nThickWidth = dlg.m_nThickWidth;

    // Update the pen used by views when drawing new strokes
    // to reflect the new pen widths for "thick" and "thin".
    ReplacePen();
}
```

6 Scroll to the top of ScribbleDoc.cpp and add the following **#include** statement:

```
#include "PenWidthsDlg.h"
```

7 Save ScribbleDoc.cpp.

When modifying ScribbleDoc.cpp, it's necessary to include PenWidthsDlg.h so that the message handler has access to the dialog class you've created.

The `OnPenWidths` function declares a `CPenWidthsDlg` object and sets the values of the `m_nThickWidth` and `m_nThinWidth` member variables to the current widths of the thick and thin pens. Then the function calls the **DoModal** function, which displays the dialog box on the screen and takes control of the application until the user exits the dialog box. If the user exits the dialog box by choosing the OK button, the function changes the current thick and thin pen widths to the new values; if the user chooses the Cancel button, the old values are retained. Finally, the function calls the `ReplacePen` member function to make the document's pen use the current widths.

When does the application perform the data exchange and validation defined in the `DoDataExchange` function? Recall that `DoDataExchange` is called by the **UpdateData** member function. Just before the dialog box is first displayed on the screen, the framework calls the **UpdateData** function with an argument of **FALSE**, which sets the contents of the edit boxes to the values of the member variables. If the user exits the dialog box by choosing the OK button, the framework calls **UpdateData** with an argument of **TRUE**, which retrieves the contents of the edit boxes and sets the values of the member variables accordingly. (If the user exits by choosing the Cancel button, the framework doesn't call **UpdateData**.)

You don't have to handle the **UPDATE_COMMAND_UI** message for the Pen Widths menu item because the menu item doesn't need to be updated. The command is never disabled since it's always legal to change the widths of the pens, and there's no need to add or remove a check mark because the command isn't a toggle.

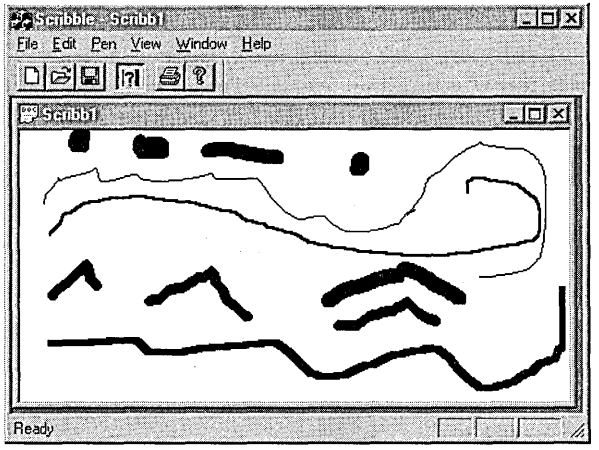
Build Scribble – Step 3 Version

How does Scribble behave now that a dialog box has been added? Compile the new version of Scribble and find out.

▶ To build Scribble — Step 3 Version

1. From the Build menu, choose Build Scribble.exe.
2. Run the new version of Scribble. Draw some strokes with the default thick pen and the default thin pen. Then use the Pen Widths dialog to change the thickness of the pens and draw some new strokes. Figure 8.6 illustrates the third version of Scribble with a variety of strokes drawn.

Figure 8.6 Scribble Version 3



3 Exit Scribble.

This completes Step 3 of the tutorial.

In the next chapter, you'll implement the updating of multiple views, scrolling, and splitter windows.

CHAPTER 9

Enhancing Views

In the previous chapters, you've seen how a view acts as an intermediary between a document and the user: The view displays a document on the screen and interprets mouse actions as operations on the document. You've also seen how a view cooperates with a frame window so that the frame window implements the generic window behavior while the view provides the application-specific functionality.

However, there are additional benefits to having a view class that is separate from the document and the frame window. This chapter describes how to take advantage of the division of labor between these classes to add special features to your application's user interface by:

- Updating multiple views on the same document.
- Scrolling a view.
- Splitting a window.

This chapter covers Step 4 of Scribble. If you want to work along, adding the code as you go, begin with the files you worked on in Chapter 8 in your Scribble project directory. At this point, these files should closely resemble those in the SCRIBBLE\STEP3 subdirectory. After following the steps outlined in this chapter, your files should closely resemble the files in the SCRIBBLE\STEP4 subdirectory.

If you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP4 subdirectory.

Note If you have not made a local copy of the sample source code for this tutorial step and you wish to do so, see "Installing the Sample Files" in Chapter 2.

You can also easily preview a running version of Scribble as it appears at the completion of this tutorial step. For more information, see "Previewing the Sample Applications" in Chapter 2.

Updating Multiple Views

To illustrate, try the following:

1 Run Scribble (your Step 3 version) and draw a few strokes in the open document.

2 From the Window menu, choose the New Window command.

This opens a new document window displaying the same drawing. The document object now has two view objects connected to it.

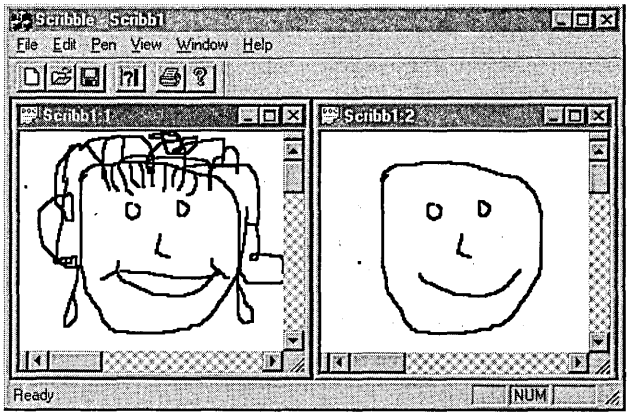
3 Choose the Tile command so you can see both views at the same time.

4 Add some more new strokes in the first window.

Do the new strokes appear in the other window simultaneously? No.

Why is this the case? Scribble, as currently implemented, has no way of telling each open document window what is happening in any other open document window. (This is illustrated in Figure 9.1.) You could force a repaint — for instance, by minimizing and then restoring the window. Then its `OnDraw` function would display the drawing again, including the new strokes. But how can you ensure that all the views attached to a document reflect changes to the document as soon as they are made?

Figure 9.1 Multiple Views on a Document Without Updating



Each view must notify the other views whenever it has modified the document. MFC provides a standard mechanism for notifying views of modifications to a document through the **UpdateAllViews** member function of the **CDocument** class.

The **UpdateAllViews** function traverses the list of views attached to the document. For each view in the list, the function calls the **OnUpdate** member function of the **CView** class. The **OnUpdate** function is where the view responds to changes in the document; the default implementation of the function invalidates the client area of the view, causing it to be repainted. The simplest way for you to use this updating mechanism in your application is to call the document's **UpdateAllViews** function whenever a view modifies a document in response to a user action.

You can also perform more efficient repainting with this updating mechanism if you use the parameters of the **UpdateAllViews** function. Here is the declaration of **UpdateAllViews**:

```
void UpdateAllViews(CView* pSender, LPARAM lHint = 0L,
                    CObject* pHint = NULL);
```

The first argument identifies the view that made the modifications to the document. This is specified to keep the **UpdateAllViews** function from performing a redundant notification; typically the view that made the modifications doesn't need to be told about them. The second two arguments are "hints." You can use these hints to describe the modifications that the view made.

The **UpdateAllViews** function gives the hints to every view attached to the document by passing them as parameters to the **OnUpdate** member function. You can override **OnUpdate** to interpret those hints and update only the area of the display that corresponds to the modified portion of the document. Thus, if another view is displaying a completely different portion of the document, it doesn't have to perform any repainting at all.

These are the basic steps you take to inform other views of modifications:

1. Define a type of hint that describes a modification to a document.
2. When a view modifies the document, create a hint describing the modification made and pass it to **UpdateAllViews**.
3. Override **OnUpdate** to use the hint so that only the portion of the screen corresponding to the modification gets updated.

These steps are described in more detail in the following sections, using Scribble as an example.

Define a Hint for Scribble

When a stroke is added to a drawing in Scribble, the rectangular region that contains the new stroke is the only area that needs to be updated; the remainder of the drawing can be left alone. Therefore, a logical choice for a hint in Scribble is the bounding rectangle of the new stroke.

Instead of creating a separate class to represent the hint, it's more convenient to pass a `CStroke` pointer as a hint. Store the bounding rectangle for each stroke in the `CStroke` object itself, so that it can be quickly referred to by `OnUpdate` to determine which area of the window needs to be repainted.

The following procedure assumes you have your Scribble project file (Scribble.mdp) open in the workspace.

▶ To define bounding rectangles for strokes

1 From ClassView, jump to the definition for class `CStroke`.

2 In the Attributes section, add the following code, just after the `m_pointArray` declaration:

```
    CRect   m_rectBounding;  // smallest rect that surrounds all
                        // of the points in the stroke
public:
    CRect& GetBoundingRect() { return m_rectBounding; }
```

The protected member variable `m_rectBounding` is a **CRect** object storing the bounding rectangle, and the public member function `GetBoundingRect` allows the rectangle to be retrieved by the view.

3 Now search in the implementation file (ScribbleDoc.cpp) for the **IMPLEMENT_SERIAL** macro, and change the schema number parameter to 2.

```
IMPLEMENT_SERIAL( CStroke, CObject, 2 )
```

This version of Scribble changes what's stored in a `CStroke` object by adding a new member variable. Changing the schema number distinguishes strokes saved by this version of Scribble from those of other versions.

4 Go to the second `CStroke` constructor (the one that initializes the pen width) and add the following line:

```
    m_rectBounding.SetRectEmpty();
```

This initializes the bounding rectangle to an empty rectangle in the constructor.

5 Jump to the `CStroke Serialize` function and add the following line just after the first **if** condition:

```
    ar << m_rectBounding;
```

This stores the `m_rectBounding` member variable in the archive.

6 Add its code pair just after the **else** branch:

```
    ar >> m_rectBounding;
```

This reads the `m_rectBounding` member variable from the archive.

In the next procedure, you'll add a helper function, `FinishStroke`. This function calculates the bounding rectangle, which is needed for smart repainting.

▶ To add the FinishStroke helper function

1 From ClassView, point your cursor at the `CStroke` class icon and click the right mouse button.

2 From the pop-up menu, choose Add Function.

The Add Member Function dialog box appears.

3 In the Function Type edit box, type the return type (in this case, `void`).

4 In the Function Declaration edit box, type the following:

```
FinishStroke()
```

5 In the Access area, select Public.

6 Choose OK.

Visual C++ adds the declaration to the header file, creates a skeleton definition in the implementation file, and jumps you to the body of the definition so you can begin typing your application-specific code.

7 Type the following code to fill in the function definition for `FinishStroke`:

```
if( m_pointArray.GetSize() == 0 )
{
    m_rectBounding.SetRectEmpty();
    return;
}
CPoint pt = m_pointArray[0];
m_rectBounding = CRect( pt.x, pt.y, pt.x, pt.y );

for (int i=1; i < m_pointArray.GetSize(); i++)
{
    // If the point lies outside of the accumulated bounding
    // rectangle, then inflate the bounding rect to include it.
    pt = m_pointArray[i];
    m_rectBounding.left   = min(m_rectBounding.left, pt.x);
    m_rectBounding.right  = max(m_rectBounding.right, pt.x);
    m_rectBounding.top    = min(m_rectBounding.top, pt.y);
    m_rectBounding.bottom = max(m_rectBounding.bottom, pt.y);
}

// Add the pen width to the bounding rectangle.  This is needed
// to account for the width of the stroke when invalidating
// the screen.
m_rectBounding.InflateRect(CSize(m_nPenWidth, m_nPenWidth));
return;
```

The `FinishStroke` member function calculates the bounding rectangle for a stroke. In this function, the stroke object iterates through its array of points, testing the location of each one; if a point falls outside the current bounding rectangle, the stroke object enlarges the bounding rectangle just enough to contain it. Then the bounding rectangle is expanded on each side by the width of the pen.

Pass the Hint After Modifying the Document

The next step is to pass the hint to the document's **UpdateAllViews** member function. It's appropriate to pass a hint each time a stroke is completed.

▶ To pass the hint after modifying the document

- From ClassView, jump to the `OnLButtonUp` member function of class `CScribbleView` and add the following code, just before the `ReleaseCapture` function call:

```
// Tell the stroke item that we're done adding points to it.
// This is so it can finish computing its bounding rectangle.
m_pStrokeCur->FinishStroke();

// Tell the other views that this stroke has been added
// so that they can invalidate this stroke's area in their
// client area.
pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);
```

The `OnLButtonUp` member function is called when a stroke is finished, so you call **UpdateAllViews** from there. In this function, the view gets the hint information that it will send to the document. It does this by calling the `FinishStroke` member function for `m_pStrokeCur`; `FinishStroke` computes the bounding rectangle for the current stroke. Then the view calls **UpdateAllViews**, passing two arguments: the **this** pointer, which identifies this view as the one that performed the modification to the document, and `m_pStrokeCur`, whose bounding rectangle is the hint. (The function sends a pointer to the entire `CStroke` object rather than just the bounding rectangle because the hint must be a **CObject** pointer, and **CRect** isn't derived from **CObject**.) The view doesn't need to send any more hint information, so it doesn't pass anything (0) in the **LPARAM** parameter.

The **UpdateAllViews** function iterates through the list of views attached to the document; for each view (except the one that performed the modification), the function calls its **OnUpdate** function and passes the hint as a parameter.

Use the Hint for Efficient Repainting

The last step is to take advantage of the hint so the other views can repaint themselves more efficiently. This involves modifying the `CScribbleView` class by overriding the `OnUpdate` function to respond to any hint it receives.

In Chapter 7, "Binding Visual Objects to Code Using WizardBar," you saw how to use WizardBar to connect user-interface objects to their message-handler functions. You can also use WizardBar to override functions inherited from the base class, which are not attached to user-interface objects. The following procedure illustrates this point.

The `OnUpdate` function for class `CScribbleView` is inherited from its base class, **CView**. `OnUpdate` appears in the WizardBar Messages list with the other virtual functions associated with the **CView** class. Messages displayed in bold have already been mapped or overridden in the class.

Note To use WizardBar to override `OnUpdate`, file ScribbleView.cpp must be open in the text editor.

▶ To add the OnUpdate function to Scribble

1 In the WizardBar Messages list box, select `OnUpdate`.

2 When prompted, choose Yes to create a handler function.

3 Fill in the skeleton function definition with the following code (you can replace the // TODO comments):

```
// The document has informed this view that some data has changed.

    if (pHint != NULL)
    {
        if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
        {
            // The hint is that a stroke has been added (or changed).
            // So, invalidate its rectangle.
            CStroke* pStroke = (CStroke*)pHint;
            CRect rectInvalid = pStroke->GetBoundingRect();
            InvalidateRect(&rectInvalid);
            return;
        }
    }
    // We can't interpret the hint, so assume that anything might
    // have been updated.
    Invalidate();
    return;
```

Recall that this function is called by the `UpdateAllViews` function of `CScribbleDoc`, which passes it a hint. In this function, the view checks if the hint is a **CStroke** object. If so, the view gets the bounding rectangle for the stroke and marks it as invalid. This rectangle marks the area that must be redrawn. If the hint isn't a **CStroke** object, the view doesn't know what area was modified, so it invalidates the entire client area as a precaution.

After a region has been invalidated, Windows sends a **WM_PAINT** message. The **OnPaint** member function defined by **CView** handles this message by calling the virtual `OnDraw` member function. Consequently, you must modify the `OnDraw` function to take advantage of the invalidated rectangle when redrawing.

4 From ClassView, jump to the `OnDraw` member function in class `CScribbleView`, and add the following code just after the `ASSERT_VALID(pDoc)` line:

```
// Get the invalidated rectangle of the view, or in the case
// of printing, the clipping region of the printer DC.
CRect rectClip;
CRect rectStroke;
pDC->GetClipBox(&rectClip);

//Note: CScrollView::OnPaint() will have already adjusted the
//viewpoint origin before calling OnDraw(), to reflect the
//currently scrolled position.
```

5 Then, add the following code immediately following the `Cstroke* pStroke = strokeList.GetNext(pos)` line:

```
rectStroke = pStroke->GetBoundingRect();
if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
    continue;
```

In the `OnDraw` function, the view first calls the **GetClipBox** member function of **CDC** to get the invalidated portion of the client area. Then the view iterates through the list of strokes in the document, calling `IntersectRect` for each to determine if any part of the stroke lies in the invalidated region. If so, the view asks the stroke to draw itself. Any strokes that don't intersect the invalidated region don't have to be redrawn.

Note This is a good point to compile your changes and test the window updating.

▶ To test your update code

1 Build and execute Scribble.

2 Add some lines to the document window.

3 From the Window menu, choose New Window, then choose Tile.

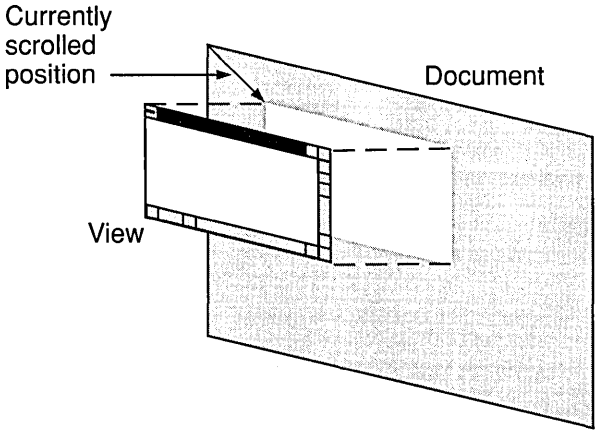
4 Draw in either window and note that the application now correctly tracks the results in both windows.

Adding Scrolling

In the current version of Scribble, you cannot work on a drawing that is larger than the window. It would be more convenient if you could work on a large drawing, no matter how small the window is. To do this, Scribble must support scrolling.

The addition of scrolling expands the conceptual role played by a view. Not only does a view produce a visual representation of a document's data, it also acts as a peephole to a document that may be too large to display all at once. This peephole can be moved across the document to reveal different portions of it. This is illustrated in Figure 9.2.

Figure 9.2 A Scrollable View on a Document



Implementing scrolling from scratch is fairly complicated. However, since a lot of the scrolling code is the same for all applications, MFC implements the common scrolling logic in a class called **CScrollView**.

Basic Steps for Adding Scrolling

The basic steps for adding scrolling to your application are as follows:

1. Define a size for your documents. This can be a constant, a member stored in each document object, a value calculated at run time, etc.
2. Derive your view class from **CScrollView** instead of **CView**.

   Note In the AppWizard–Step 6 dialog box, you have the option of changing your base class. You could have, for example, chosen **CScrollView** instead of **CView** at that point, thereby eliminating some of the steps in this procedure.
3. Pass the document's size to the **SetScrollSizes** member function of **CScrollView** whenever the size may change.
4. Convert between logical coordinates and device coordinates if passing points between graphic device interface (GDI) and non-GDI functions.

The framework's responsibilities are as follows:

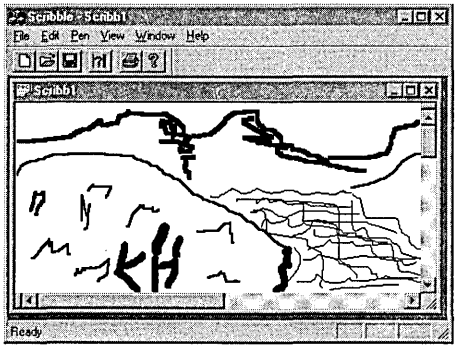
- Handle all **WM_HSCROLL** and **WM_VSCROLL** messages, scroll the document in response, and move the scroll box accordingly.

  The positions of the scroll boxes reflect where the currently displayed portion of the document resides relative to the rest of the document. If the user clicks on a scroll arrow at either end of the scroll bar, the document scrolls one "line" (whose meaning depends on the document type). If the user clicks on either side of the scroll box, the document is scrolled one page. If the user drags the scroll box itself, the document is scrolled accordingly.

- Calculate a mapping between the lengths of the scroll bars and the height and width of the document, adjust this scaling factor when the window is resized or when the size of the document changes, and in turn remove or add scroll bars as needed.

The next section describes how to add scrolling to Scribble. Figure 9.3 shows what Scribble looks like with scroll bars added.

Figure 9.3 Scribble with Scrolling Support



Add Scrolling to Scribble

The following procedures describe how to perform the first three steps involved in adding scrolling to Scribble, as described in "Basic Steps for Adding Scrolling." In the section, "Working with GDI Coordinates," you'll see how to perform the fourth step.

▶ To add scrolling support to Scribble

1 From ClassView, jump to the definition of class `CScribbleDoc`, and add the following code after the **public** Attributes section:

```
protected:
    CSize m_sizeDoc;
public:
    CSize GetDocSize() { return m_sizeDoc; }
```

This defines the size of Scribble documents by having each document store its dimensions. The member variable `m_sizeDoc` stores the size of the document in a **CSize** object. This member is protected, so it cannot be accessed directly by the views attached to the document. To let the views retrieve the size of the document, you provide a public helper function named `GetDocSize`. The views base their scrolling limits on the document size.

2 Jump to the `Serialize` member function in `CScribbleDoc` and add the following line of code in place of the \\TODO comments for storing:

```
ar << m_sizeDoc;
```

3 Add the matching line just after the **else** branch:

```
ar >> m_sizeDoc;
```

4 Jump to the `InitDocument` function, and add the following code after the call to `ReplacePen`:

```
// Default document size is 800 x 900 screen pixels.
m_sizeDoc = CSize(800,900);
```

The new code in the `InitDocument` member function initializes the `m_sizeDoc` member variable; recall that you use this function whenever a new document is created or an existing document is opened. All Scribble documents are the same size: 800 logical units in width and 900 logical units in height. For simplicity's sake, Scribble doesn't support documents of varying size.

The changes to the `Serialize` member function store and read the `m_sizeDoc` member variable.

5 Jump to the `CScribbleView` class definition and specify that it be derived from class **CScrollView** (instead of class **CView**):

```
class CScribbleView : public CScrollView
```

Recall that MFC uses message maps as well as C++ inheritance. As a result, modifying the class declaration in the header file isn't enough to give `CScribbleView` all of **CScrollView**'s functionality. You also have to modify the message-map macros in the implementation file.

6 In ScribbleView.cpp, change the reference to **CView** in the following lines to refer to **CScrollView** instead:

```
IMPLEMENT_DYNCREATE( CScribbleView, CScrollView )
BEGIN_MESSAGE_MAP( CScribbleView, CScrollView )
```

In the message map macro, referencing **CScrollView** instead of **CView** instructs the framework to search **CScrollView**'s message map if it can't find the message handler it needs in `CScribbleView`'s message map.

7 If you want to use the diagnostic features provided by MFC, change the implementations of the `Dump` and `AssertValid` member functions of `CScribbleView`. These functions simply call their base class versions; change them to call the **CScrollView** versions rather than the **CView** versions.

These changes set the document's scrolling limits according to the size of the document. By changing the base class of `CScribbleView` from **CView** to **CScrollView**, you give `CScribbleView` scrolling functionality without having to implement scrolling yourself.

Since Scribble documents are fixed in size, there is no need to make any subsequent calls to **SetScrollSizes**. If your application supports documents of varying size, you should call **SetScrollSizes** immediately after the document's size changes. (You can do this from the **OnUpdate** member function of your view class.)

In addition to the changes just made, the `CScribbleView` class will override the **OnInitialUpdate** member function, which is called when the view is first attached to the document. By overriding this function, you can inform the view of the document's size as soon as possible.

The following procedure describes how to do this.

Note To use the WizardBar for this procedure, first open file ScribbleView.cpp in the text editor.

▶ To override OnInitialUpdate

1. In the WizardBar Messages list box, select `OnInitialUpdate`.
2. When prompted, choose Yes to add a function handler.
3. Add the following line of code just before the call to the `OnInitialUpdate` function in the base class:

```
SetScrollSizes( MM_TEXT, GetDocument()->GetDocSize() );
```

The **SetScrollSizes** member function is defined by **CScrollView**. Its first parameter is the mapping mode used to display the document. The current version of Scribble uses **MM_TEXT** as the mapping mode; in Chapter 10, Scribble will use the **MM_LOENGLISH** mapping mode for better printing. (For more information on mapping modes, see "Enlarge the Printed Image" in Chapter 10, or see **CDC::SetMapMode** in the *Class Library Reference*).

The second parameter is the total size of the document, which is needed to determine the scrolling limits. The view uses the value returned by the document's `GetDocSize` member function for this parameter.

SetScrollSizes also has two other parameters for which Scribble uses the default values. These are **CSize** values that represent the size of one "page" and one "line," the distances to be scrolled if the user clicks the scroll bar or a scroll arrow. The default values are 1/10th and 1/100th of the document size, respectively.

Working with GDI Coordinates

Notice that the addition of scrolling didn't require you to modify the `OnDraw` member function of `CScribbleView`. If the drawing function is unchanged, why does the window display different portions of the document depending on where the user has scrolled to? The reason is that the document is displayed using coordinates relative to an origin used by GDI. When this origin was fixed at the upper-left corner of the client area, the part of the document that was visible was always the same. By moving the origin used by GDI, **CScrollView** can adjust which portion of the document is shown in the client area of the window and which portions are hidden.

The origin used by GDI is a characteristic of a device context; it is used by the member functions of the **CDC** class. If you want to make adjustments to the **CDC** object used by your view, you can override the **OnPrepareDC** member function

defined by **CView**. **CScrollView** overrides **OnPrepareDC** to move the device context's origin to reflect the currently scrolled position. **OnPrepareDC** is always called by the framework before it calls **OnDraw**. As a result, you don't have to make any changes to the `OnDraw` function to draw a properly scrolled document; all the work needed to do scrolling is done to the device context before `OnDraw` receives it.

It's important to note that changing the device context's origin doesn't affect the coordinates you receive with Windows messages such as **WM_LBUTTONDOWN** or **WM_MOUSEMOVE**; the points accompanying those messages are still specified in coordinates relative to the upper-left corner of the client area. This is because Windows messages are not part of a device context, so they are unaffected by changes to the GDI origin. Thus, `CScribbleView` must now deal with two types of coordinates:

- The coordinates used for describing the points received with a mouse message. Those points are returned in "device coordinates."
- The coordinates used for drawing with GDI. These are known as "logical coordinates."

Converting from Device Coordinates to Logical Coordinates

When storing the coordinates of strokes, Scribble needs to know where the strokes are relative to the document, not relative to the client area. Consequently, `CScribbleView` must convert points from device coordinates (relative to the window origin) to logical coordinates (relative to the document origin) before storing them in `CStroke` objects.

▶ To store the strokes using logical coordinates

1 Use ClassView to jump to the `OnLButtonDown` member function of class `CScribbleView`, and add the following code to the beginning of the function definition:

```
// CScrollView changes the viewport origin and mapping mode.
// It's necessary to convert the point from device coordinates
// to logical coordinates, such as are stored in the document.
CClientDC dc(this);
OnPrepareDC(&dc);
dc.DPtoLP(&point);
```

In this function, the view receives a point specified in device coordinates. A device context is needed to find the GDI origin, so the function declares a **CClientDC** object, which is a **CDC** object for the client area of the view, and calls **OnPrepareDC** to adjust its origin. Then the function passes the point to the **DPtoLP** (Device Point to Logical Point) member function of **CDC** to perform the actual conversion. The point added to `m_pStrokeCur` is thus described in logical coordinates (that is, relative to the document origin).

2 Jump to `OnMouseMove` and add similar code just after the line `CClientDC dc(this);`:

```
// CScrollView changes the viewport origin and mapping mode.
// It's necessary to convert the point from device coordinates
// to logical coordinates, such as are stored in the document.
OnPrepareDC(&dc);
dc.DPtoLP(&point);
```

This function already has a device context for drawing the stroke in progress, so the only modifications needed are to call **OnPrepareDC** to move the viewport origin and then **DPtoLP** to convert the point before adding it.

3 Make the same modification to the `OnLButtonUp` member function, again, just after the line `CClientDC dc(this);`:

```
// CScrollView changes the viewport origin and mapping mode.
// It's necessary to convert the point from device coordinates
// to logical coordinates, such as are stored in the document.
OnPrepareDC(&dc);  // set up mapping mode and viewport origin
dc.DPtoLP(&point);
```

Like `OnMouseMove`, this function already has a device context to complete drawing the stroke, so the only modifications needed are to call **OnPrepareDC** and then **DPtoLP**.

4 Jump to `OnUpdate` and add the following lines of code just after the line `CStroke* pStroke = (CStroke*)pHint;`:

```
CClientDC dc(this);
OnPrepareDC(&dc);
```

5 Skip a line (the `Crect rectInvalid` ... line) and add the following line:

```
dc.LPtoDP(&rectInvalid);
```

Unlike the previous three functions, `OnUpdate` requires a conversion in the opposite direction; that is, from logical coordinates to device coordinates. Recall that `OnUpdate` retrieves the bounding rectangle of a stroke and invalidates that rectangle. The stroke's bounding rectangle is stored in logical coordinates. However, the rectangle passed to **InvalidateRect** must be specified in device coordinates (since **InvalidateRect** is not a GDI function). Accordingly, a stroke's bounding rectangle must have its coordinates converted into device coordinates before it can be invalidated.

The function declares a **CClientDC** object and then calls the **OnPrepareDC** member function to move the viewport origin of the device context to reflect the currently scrolled position. The rectangle is then passed to the **LPtoDP** (Logical Point to Device Point) function of **CDC** to convert its points into device coordinates. (Both **DPtoLP** and **LPtoDP** are overloaded to accept rectangles as well as points.) Once it is converted, the rectangle can be invalidated.

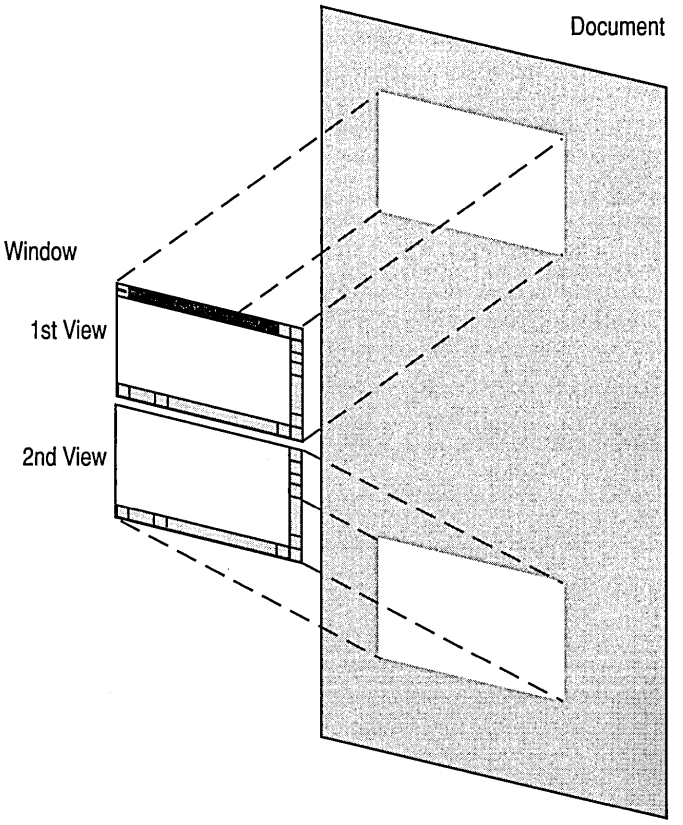
For more information on **CScrollView**, see the *Class Library Reference*.

Note This is a good point to compile and test Scribble's scrolling.

Adding Splitter Windows

Scrolling lets you work on a document that is larger than the window, but by the same token it means that much of the document is hidden at any one time. Suppose the user needs to refer to two widely separated portions of a document at the same time. One way to do this is to open another window on the same document and scroll both window to different locations. However, windows must be resized individually so that they don't overlap. A more convenient solution is to divide a window into separate "panes," each of which can display a different portion of the document. This is illustrated in Figure 9.4.

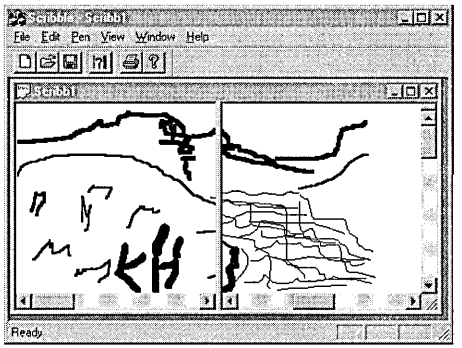
Figure 9.4 A Window with Two Views on a Document



A window that can be divided into multiple panes is called a "splitter window." A splitter window contains split boxes at the top of the vertical scroll bar and at the left of the horizontal scroll bar. By double-clicking a split box, the user can divide a window vertically or horizontally into panes. The panes are separated by a "split bar"; each pane can be scrolled independently to display a different portion of the document. The user can also drag the split bar to resize both panes at once.

Figure 9.5 shows what a Scribble window looks like when it is split into two panes.

Figure 9.5 Scribble Document Window Split into Two Panes



Each pane in a splitter window represents a separate view object. In Figure 9.5, each pane is an instance of the `CScribbleView` class, but it's not necessary for the panes to use the same view class; you can use different classes for different panes. This is useful when, for example, you want one pane to display an outline of a document while the other pane displays the full text.

How the Framework Supports Splitting

For an MDI application such as Scribble to support splitting, objects of three classes must cooperate to display a document: a **CMDIChildWnd** object, which manages the document window's frame; a **CSplitterWnd** object, which manages the document window's client area; and one or more `CScribbleView` objects, each of which manages a pane in the window. The **CSplitterWnd** object is not visible as a distinct entity, but it is responsible for handling the `CScribbleView` objects as panes, managing their scroll bars, and drawing the split boxes and split bars.

This technique for managing splitter windows is similar to the implementation of MDI in general. A client window manages the entire client area, or workspace, of an MDI application's frame window. It is this client window that owns the child windows that display documents.

Because you specified Scribble as an MDI application, AppWizard creates the **CChildFrame** class, derived from **CMDIChildWnd**. You'll add the code to support splitter windows in Scribble to this class. If Scribble were an SDI application, you would add the splitter window functionality directly to the **CMainFrame** class.

There are two ways that you can add splitter window functionality to your application:

- You can choose the Use Splitter Window option in AppWizard when you first create the application's skeleton files. This method performs the steps outlined below for you.
- You can add this functionality "manually," using ClassWizard to automate the process somewhat.

You'll use the manual method for Scribble because it demonstrates how simply the framework implements this feature. The method for using AppWizard is described later in this section, in the topic "Adding Splitter Window Functionality by Using AppWizard."

Basic Steps to Add Splitter Windows

By performing the following steps, you can easily add splitter windows to your applications:

1. Derive a frame window class from **CMDIChildWnd** if you are writing a Multiple Document Interface (MDI) application.

   Because you specified Scribble as an MDI application, AppWizard generated a **CChildFrame** class for you, derived from **CMDIChildWnd**.

2. Give this class a member variable of type **CSplitterWnd**. (For SDI applications, add the member variable to your **CMainFrame** class.)

   This is the window that covers the frame window's client area.

3. Override the **OnCreateClient** member function of your frame window class to create a **CSplitterWnd**. (For SDI applications, the frame window class is **CMainFrame**.)

   The framework calls this function when it first creates the frame window. For Scribble, you will override the **OnCreateClient** member function of **CChildFrame**.

Add Splitter Windows to Scribble

Adding splitter windows to Scribble requires only two very simple steps. As shown in the following procedures, you'll add a member variable, `m_WndSplitter`, and a function, `OnCreateClient`, to class `CChildFrame`. Once this is done, you can build and run Scribble to test the splitter window functionality.

Note When you choose the splitter window option from AppWizard, AppWizard also generates a menu item, which the user can use if they like instead of directly selecting the split box. To duplicate this feature, create a Split item on Scribble's Window menu, and assign it an ID of **ID_WINDOW_SPLIT**. There's no need to create a handler for it—the framework handles it automatically.

▶ To declare the CSplitterWnd member variable

1 In ClassView, point to the `CChildFrame` class icon and click the right mouse button.

2 From the pop-up menu, choose Add Variable.

The Add Member Variable dialog box appears.

3 In the Variable Type edit box, type: `CSpitterWnd`

4 In the Variable Declaration edit box, type: `m_wndSplitter`

5 In the Access area, specify "protected."

6 Choose OK.

You can view the new variable in ClassView under the `CChildFrame` class.

▶ To add the OnCreateClient member function

1 Open ChildFrm.cpp in the editor window.

2 From the WizardBar Messages list, choose OnCreateClient.

3 When prompted, choose Yes to create the handler.

4 Within the skeleton function definition that ClassWizard creates, type the following (replace the existing code or comments):

```
return m_wndSplitter.Create(this,
        2, 2,       // TODO: adjust the number of rows, columns
        CSize(10, 10),  // TODO: adjust the minimum pane size
        pContext);
```

5 If you like, save the header and implementation files for `CChildFrame`.

You can view the new member function in ClassView, under class `CChildFrame`.

In the `OnCreateClient` member function, the frame window creates the window that will cover its client area by calling the **Create** function of its **CSplitterWnd** member variable. The parameters passed to the **Create** function describe the panes that the splitter window will manage.

The first argument passed to **Create** specifies the parent window for the client window: The function passes the **this** pointer, making the `CScribbleFrame` window the parent of the **CSplitterWnd** object (the member variable, `m_wndSplitter`). The second and third parameters specify the maximum number of rows and columns that the splitter window can have; a value of two is used for each, so Scribble's splitter windows can have up to four panes. The fourth parameter specifies the minimum size of a pane: a square 10 logical units on a side. The fifth parameter is the **CCreateContext** structure that is passed to `OnCreateClient`. This structure is used to determine which view class should be used for each pane in the splitter window.

The **Create** function can also accept an additional two arguments; because Scribble doesn't pass any values for these, the default values are used. The sixth argument specifies the styles to be used for the splitter window. The default value specifies a visible child window with vertical and horizontal scroll bars that supports dynamic splitting. The seventh argument specifies the ID to be assigned to the splitter window. Its default value is **AFX_IDW_PANE_FIRST**, which is the ID of the first pane.

Adding Splitter Window Functionality by Using AppWizard

The previous section described how to add splitter window functionality to an MDI application that you had already developed. By performing the following procedure, AppWizard adds the splitter window functionality for you when you first create a project.

Note This procedure isn't relevant to the version of Scribble you've developed to this point, as it requires starting over with AppWizard. It is included here for information only.

▶ To add splitter windows by using AppWizard

1. Start AppWizard and specify your preferences in the first three dialog boxes.
2. In the AppWizard Step 4 dialog box, choose the Advanced button.
3. Select the Window Styles tab.
4. Select the Use Splitter Window option, and choose Close.
5. Finish choosing your options in AppWizard, and choose OK in the New Project Information dialog box.

For more information on **CSplitterWnd**, see the *Class Library Reference*.

Build Scribble – Step 4 Version

How does Scribble behave with these new enhancements? Build the new version and find out.

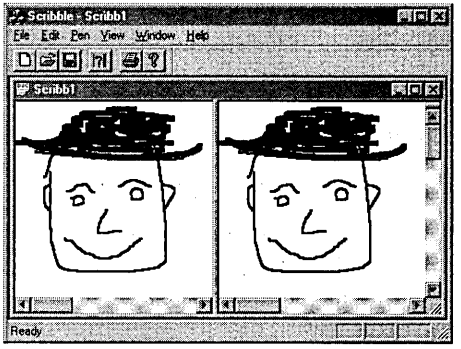
▶ To build Scribble

- From the Build menu, choose Build Scribble.exe.

Run the new version of Scribble.

Draw some strokes, scroll to a new portion of the drawing, and draw some more strokes. Resize the window and scroll back and forth. Double-click the split box, or drag it to split the window into two panes. With both panes displaying the same portion of the document, draw some strokes in one pane and see them reflected in the other one. Figure 9.6 shows this version of Scribble.

Figure 9.6 Scribble Version 4



Exit Scribble.

This completes Step 4 of the tutorial. You now have a basic understanding of the view architecture provided by the Microsoft Foundation Class Library.

In the next chapter, you'll enhance Scribble's printing and print preview support.

CHAPTER 10

Enhancing Printing

Scribble has supported printing and print preview since Chapter 5, "Creating the View," when you first added application-specific code to the starter files created by AppWizard. All the printing and previewing functionality came for free. None of the code you added dealt specifically with printing; AppWizard and the framework did all the work for you.

While it's nice to get printing and print preview for free, Scribble's current printing support isn't perfect. For example, the printed image is smaller than you might like. In addition, the printed image is very plain; it doesn't include a header or footer. This chapter describes how to enlarge the printed image and implement printing enhancements in your application.

This chapter covers the following topics:

- Enhance Scribble's printing
- Enhance Scribble's print preview

This chapter covers Step 5 of Scribble. If you want to work along, adding the code as you go, begin with the files in your Scribble project directory. At this point, these files should closely resemble those in the SCRIBBLE\STEP4 subdirectory. After following the steps outlined in this chapter, your files should closely resemble the files in the SCRIBBLE\STEP5 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP5 subdirectory.

Note If you have not made a local copy of the sample source code for this tutorial step and you wish to do so, see "Installing the Sample Files," in Chapter 2.

You can also easily preview a running version of Scribble as it appears at the completion of this tutorial step. For more information, see "Previewing the Sample Applications," in Chapter 2.

For more information on the framework's printing architecture, see the article "Printing" in *Programming with MFC*.

Enhance Scribble's Printing

Step 5 of Scribble adds the following printing capabilities to the program:

- Enlarges the printed image by using the **MM_LOENGLISH** mapping mode
- Paginates a Scribble document
- Adds a page header

These enhancements will invalidate previous Scribble file formats. To alleviate this issue, you'll first increment Scribble's serialization.

The following topics describe these enhancements in detail.

Enlarge the Printed Image

Recall from "Add Scrolling to Scribble," in Chapter 9, that when you specify a position for a GDI drawing function, you use logical coordinates. Chapter 9 described how **CScrollView** moves the origin of this coordinate system. You can also control the scale of this coordinate system, that is, the physical size of a logical unit. By default, GDI considers logical units to be equal to device units, meaning that 1 logical unit equals 1 pixel on the screen. This interpretation of logical units is called **MM_TEXT** mapping mode.

Since Scribble uses **MM_TEXT** mapping mode, it considers a stroke that is 100 units long to be 100 pixels long. The physical size of the stroke depends on the device that displays it. For example, a device unit on a typical laser printer is 1/300 of an inch, which is considerably smaller than a pixel on a typical screen. As a result, the images that Scribble produces on a printer are much smaller than those it displays on the screen.

To keep Scribble from producing tiny images on the printer, you need a mapping mode that ensures that a drawing remains the same size no matter what device displays it. Windows provides several such mapping modes, known as metric mapping modes. In these modes, GDI considers logical units to be equal to real-world units (or metrics), such as millimeters or inches.

Using the MM_LOENGLISH Mapping Mode

In Step 5, Scribble changes to **MM_LOENGLISH** mapping mode, which treats each logical unit as 0.01 inches. In this mode, a stroke that is 100 logical units long is drawn as 1 inch long, no matter which device is used; each device driver determines how many device units are needed to draw a 1-inch stroke.

Once Scribble uses **MM_LOENGLISH** mode, all coordinates used for GDI drawing are in hundredths of an inch, not pixels. As a result, the images that Scribble displays on the printer are the same size as the ones it displays on the screen. Recall that in Chapter 9, a Scribble drawing was defined to be 800 logical units across and 900 logical units high; once you change the mapping mode, a drawing is 8 inches across and 9 inches high.

Another feature of **MM_LOENGLISH** mode (as well as the other metric mapping modes) is that its y-axis runs in the opposite direction to that in **MM_TEXT** mode. In **MM_TEXT** mode, y-coordinates increase when you move down, but in all the metric mapping modes, y-coordinates increase when you move up.

After you change the mapping mode, you need to make several adjustments to the existing Scribble code. These changes are described in the procedure "To compensate for the reversal of the y-axis." For an explanation of the modifications, see "Compensating for the Reversal of the Y-Axis."

Serializing Scribble's New Stroke Information

All of these code changes, including the change to the mapping mode, invalidate previous Scribble file formats. For example, the x and y coordinates of earlier Scribble drawings were based on pixel locations; now they are interpreted according to 1/100ths of an inch. Also, because of the metric mapping mode, the sign of the y coordinate reverses, and what used to be a positive value is now negative. To compensate for this reversal, you'll also modify the `m_rectBounding` variable, which is a member of `CStroke`, the class that stores Scribble's stroke data.

What does this mean? If the new version of Scribble tries to read in stroke data from an older version of Scribble, it will still try to draw the stroke, but the old stroke data might map to an area, for example, outside the current window.

Recall that, when you defined a hint for Scribble, you modified the way Scribble stored strokes by adding the member variable, `m_rectBounding`. In that tutorial step, you incremented the schema number of the **IMPLEMENT_SERIAL** macro to distinguish between stroke data from different versions of Scribble. Because of the code changes you'll make in following sections, you need to increment this schema number again.

Note After marking the version of Scribble data in this way, you'll receive an error message "Unrecognized file format" if you try to load drawings from previous versions of Scribble. Naturally, this is preferable to opening a drawing and seeing nothing on the screen.

▶ To increment Scribble's serialization

- Open ScribbleDoc.cpp, search for the **IMPLEMENT_SERIAL** macro, and change the schema number parameter to 3.

```
IMPLEMENT_SERIAL( CStroke, CObject, 3 )
```

The next topic describes how to specify **MM_LOENGLISH** as the new mapping mode.

Specify the Mapping Mode

You specify the mapping mode in Scribble when you call the **SetScrollSizes** member function defined by **CScrollView**. Recall from Chapter 9, that this function sets the view's scrolling limits. **SetScrollSizes** is called from the `OnInitialUpdate` member function of `CScribbleView`.

▶ To specify the mapping mode

1 Use ClassView or WizardBar to jump to the `OnInitialUpdate` function in class `CScribbleView`.

2 Replace **MM_TEXT** with **MM_LOENGLISH** as illustrated below.

```
SetScrollSizes( MM_LOENGLISH, GetDocument()->GetDocSize() );
```

Recall that `OnInitialUpdate` is called immediately after the view is attached to the document. This lets the view set its mapping mode before `OnDraw` is called.

Compensating for the Reversal of the Y-Axis

Even though **MM_LOENGLISH** mapping has changed the direction of the y-axis for drawing, most of the current Scribble code doesn't require any modifications. This is because the **DPtoLP** function performs the conversion for you. (**DPtoLP** is called by the mouse event handler functions: **OnLButtonDown**, **OnMouseMove**, and **OnLButtonUp**.)

Consider this: when a point is received with a mouse message, its coordinates are converted by the **DPtoLP** function before being stored in a `CStroke` object. This means its y-coordinates are converted from a positive number of pixels to a negative number of inches (1 pixel = .01 inch). Those coordinates are then passed to the **LineTo** drawing function, and then it's up to the device driver for the screen to determine how many pixels are equivalent to the value that was passed in inches. You never have to directly examine the value of the coordinates.

However, there are some places where the reversal of the y-axis does have an impact. The mapping mode used by GDI is a characteristic of a device context; functions that don't use a device context are unaffected by the mapping mode. The member functions of the **CRect** class don't use the mapping mode; consequently, you must make some adjustments wherever Scribble uses **CRect** functions.

▶ To compensate for the reversal of the y-axis

1 From ClassView, jump to the `FinishStroke` member function definition of class `CStroke`.

2 Modify the code, as shown below, by reversing the **min** and **max** functions for the top and bottom of the bounding rectangle, and adding a cast to the y-axis `m_nPenWidth` parameter. These modifications take into account the negative sign of the y coordinates.

Note that the lines of code shown already exist—you are just modifying them slightly, as described.

```
void CStroke::FinishStroke()
{
    // ...

        m_rectBounding.top      = max(m_rectBounding.top, pt.y);
        m_rectBounding.bottom   = min(m_rectBounding.bottom, pt.y);
    }

    // ...
    m_rectBounding.InflateRect
        (CSize(m_nPenWidth,-(int)m_nPenWidth));
    return;
}
```

You must also make a correction when using the invalid rectangle. Recall that the `OnDraw` member function checks whether the invalid rectangle intersects the bounding rectangle for each stroke. The **IntersectRect** member function of **CRect** assumes that the bottom of a rectangle must have a larger y-coordinate than that of the top; it cannot find the intersection of two rectangles whose bottoms have smaller y-coordinates than their tops.

3 Use ClassView to jump to the `OnDraw` member function of `CScribbleView`, and add the following lines of code after the line `pDC->GetClipBox(&rectClip);`:

```
pDC->LPtoDP(&rectClip);
rectClip.InflateRect(1, 1); // avoid rounding to nothing
```

4 Add the following lines of code after the line `rectStroke = pStroke->GetBoundingRect();`:

```
pDC->LPtoDP(&rectStroke);
rectStroke.InflateRect(1, 1);
```

Both the invalidated rectangle and the bounding rectangle are converted to device coordinates (changing the signs of the coordinates to positive) before being tested for intersection. They are also inflated by one pixel in case they were rounded to nothing during the conversion.

Paginate Scribble Documents

If Scribble allowed you to produce arbitrarily large drawings, it would make sense for the program to break up a drawing into pages by dividing it into a grid of *m* by *n* rectangles, the values of *m* and *n* being determined by the size of the drawing. However, Scribble supports drawings of only one size, and each one fits on a single page. To illustrate pagination, Step 5 of Scribble prints each drawing as a two-page document: a title page, and the drawing itself.

Adding Pagination

To add pagination, you'll:

- Modify the OnPreparePrinting member function
- Override the default OnPrint member function
- Add two new helper functions: `PrintTitlePage`, which prints the title page, and `PrintPageHeader`, which prints a header on the drawing page.

▶ To modify OnPreparePrinting

- Use ClassView or WizardBar to jump to the `OnPreparePrinting` function definition of class `CScribbleView`, and add the following code (replace the existing comment and return code):

```
pInfo->SetMaxPage(2);  // the document is two pages long:
                       // the first page is the title page
                       // the second page is the drawing
```

This function specifies the length of the document by calling **SetMaxPage** for the *pInfo* parameter. Since all Scribble documents are two pages long, the function uses a numeric constant rather than a variable to represent the number of the last page of the document. The title page and the drawing page are numbered 1 and 2, respectively. Later, you'll add a Scribble-specific version of the call to **DoPreparePrinting** that AppWizard generated (which you just replaced), when you enhance Scribble's Print Preview. This function displays the Print dialog box and creates a device context for the printer.

You've used WizardBar before to override base class functions. The following procedure assumes you have file ScribbleView.cpp open in the editor.

▶ To override OnPrint

1. In the WizardBar Messages list, choose **OnPrint**.
2. When prompted, respond Yes to create a handler.

3 Replace the highlighted comment and code generated by WizardBar with the following:

```
if (pInfo->m_nCurPage == 1)   // page no. 1 is the title page
{
    PrintTitlePage(pDC, pInfo);
}
else
{
    CString strHeader = GetDocument()->GetTitle();

    PrintPageHeader(pDC, pInfo, strHeader);
    // PrintPageHeader() subtracts out from the pInfo->m_rectDraw
the
    // amount of the page used for the header.
    pDC->SetWindowOrg(pInfo->m_rectDraw.left,-pInfo-
>m_rectDraw.top);

    // Now print the rest of the page
    OnDraw(pDC);
}
return;
```

The behavior of the `OnPrint` member function depends on which of the two pages is being printed. If the title page is being printed, `OnPrint` simply calls the `PrintTitlePage` function and then returns. If it's the drawing page, `OnPrint` calls `PrintPageHeader` to print the header and then calls `OnDraw` to do the actual drawing. Before calling `OnDraw`, `OnPrint` sets the window origin at the upper-left corner of the rectangle defined by **m_rectDraw**; this rectangle was reduced by `PrintPageHeader` to account for the size of the header. This keeps the drawing from overlapping the header.

Notice that the drawing itself isn't divided into multiple pages. Consequently, `OnDraw` never has to display just a portion of the drawing (for example, it never has to display the section that fits on a particular page without displaying the surrounding sections). Either the title page is being printed and `OnDraw` isn't called at all, or else the drawing page is being printed and `OnDraw` displays the entire drawing at once.

This also explains why `CScribbleView` doesn't override the **OnPrepareDC** member function: there's no need to adjust the viewport origin or clipping region depending on which page is being printed.

Adding the Helper Functions

The next step is to add the new helper functions. By using the Add Function pop-up menu command, you can declare and define them in one step.

As mentioned, the `PrintTitlePage` function prints a title page, and `PrintPageHeader` prints a header on the drawing page.

▶ To add the PrintTitlePage helper function

1 In the ClassView, point to `CScribbleView` and click the right mouse button.

2 From the pop-up menu, choose the Add Function command.

3 In the Function Type edit box, type the return type of the function (in this case, `void`).

4 In the Function Declaration edit box, type the following:

```
PrintTitlePage(CDC* pDC, CPrintInfo* pInfo)
```

5 In the Access area, select Public, and choose OK.

ClassWizard adds the declaration to the Public section of the header file, creates a skeleton definition in the implementation file, and jumps you to the body of the definition so you can begin typing your application-specific code.

6 Implement `PrintTitlePage` with the following code:

```
// Prepare a font size for displaying the file name
LOGFONT logFont;
memset(&logFont, 0, sizeof(LOGFONT));
logFont.lfHeight = 75; //  3/4th inch high in MM_LOENGLISH
CFont font;
CFont* pOldFont = NULL;
if (font.CreateFontIndirect(&logFont))
    pOldFont = pDC->SelectObject(&font);

// Get the file name, to be displayed on title page
CString strPageTitle = GetDocument()->GetTitle();

// Display the file name 1 inch below top of the page,
// centered horizontally
pDC->SetTextAlign(TA_CENTER);
pDC->TextOut(pInfo->m_rectDraw.right/2, -100, strPageTitle);

if (pOldFont != NULL)
    pDC->SelectObject(pOldFont);
```

The `PrintTitlePage` function uses **m_rectDraw**, which stores the usable drawing area of the page, as the rectangle in which the title should be centered.

Notice that `PrintTitlePage` declares a local **CFont** object to use when printing the title page. If you needed the font for the entire printing process, you could declare a **CFont** member variable in your view class, create the font in the **OnBeginPrinting**, and destroy it in **EndPrinting**. However, since Scribble uses the font for just the title page, the font doesn't have to exist beyond the `PrintTitlePage` function. When the function ends, the destructor is automatically called for the local **CFont** object.

▶ To add the PrintPageHeader helper function

1 From the ClassView pop-up menu, choose the Add Function command. (As in the previous procedure, your mouse cursor should be pointing at class `CScribbleView` when you invoke the pop-up menu.)

The Add Function dialog box appears.

2 In the Function Type edit box, type the return type of the function (in this case, `void`).

3 In the Function Declaration edit box, type the following:

```
PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
                CString& strHeader)
```

4 In the Access area, select Public, and choose OK.

5 Add the following code to the function body:

```
// Specify left text alignment
pDC->SetTextAlign(TA_LEFT);

// Print a page header consisting of the name of
// the document and a horizontal line
pDC->TextOut(0,-25, strHeader);  // 1/4 inch down

// Draw a line across the page, below the header
TEXTMETRIC textMetric;
pDC->GetTextMetrics(&textMetric);
int y = -35 - textMetric.tmHeight;     // line 1/10th in.
                                       // below text
pDC->MoveTo(0, y);                     // from left margin
pDC->LineTo(pInfo->m_rectDraw.right, y); //  to right margin

// Subtract from the drawing rectangle the space used by header.
y -= 25;    // space 1/4 inch below (top of) line
pInfo->m_rectDraw.top += y;
```

The `PrintPageHeader` member function prints the name of the document at the top of the page, and then draws a horizontal line separating the header from the drawing. It adjusts the **m_rectDraw** member of the *pInfo* parameter to account for the height of the header; recall that `OnPrint` uses this value to adjust the window origin before it calls `OnDraw`.

Enhance Scribble's Print Preview

The default print preview capabilities are almost sufficient for Scribble's needs. To some extent, Scribble's print preview has already been enhanced when the printing capabilities were enhanced. Recall that in the override of `OnPreparePrinting` you called the **SetMaxPages** function to specify the length of Scribble documents. This allows the framework to add a scroll bar to the preview window.

Another enhancement you can make is to change the number of pages displayed when preview mode is invoked.

For more information on the framework's print preview architecture, see the article "Printing" in *Programming with MFC*.

▶ To set the number of pages displayed in preview mode

- Use ClassView or WizardBar to jump to the `OnPreparePrinting` member function in ScribbleView.cpp. Then add the following code to the end of the function:

```
BOOL bRet = DoPreparePrinting (pInfo);   // default preparation
pInfo->m_nNumPreviewPages = 2;   //Preview 2 pages at a time
// Set this value after calling DoPreparePrinting to override
// value read from .INI file
return bRet;
```

The lines added here assign the value 2 to **m_nNumPreviewPages**. This causes Scribble to preview both pages of the document at once: the title page (page 1) and the drawing page (page 2). Note the value for **m_nNumPreviewPages** must be assigned after calling **DoPreparePrinting**, because **DoPreparePrinting** sets **m_nNumPreviewPages** to the number of preview pages used the last time the program was executed; this value is stored in the application's .INI file.

Compile Scribble – Step 5 Version

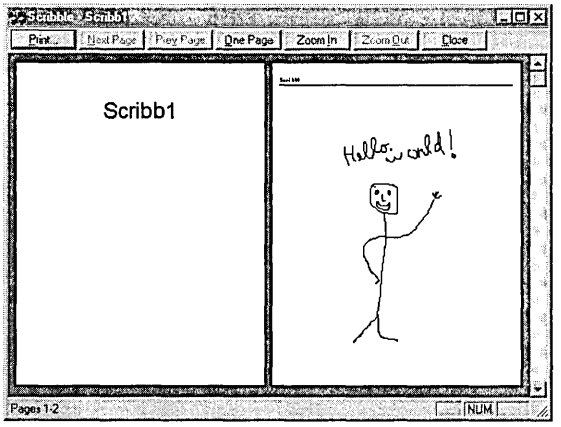
What does Scribble's printing look like now? Build the new version of Scribble and find out.

▶ To build Scribble

- From the Build menu, choose Build Scribble.exe.

Run the new version of Scribble. Draw some strokes, and then choose the Print Preview from the File menu. Switch back and forth between one-page and two-page display mode, or move to the previous or next page. Figure 10.1 shows this version of Scribble.

Figure 10.1 Scribble Version 5



Exit Scribble.

This completes Step 5 of the Scribble tutorial. For a deeper understanding of the printing architecture provided by MFC, see the article "Printing" in *Programming with MFC*.

In the next chapter, you'll add context-sensitive help to Scribble.

CHAPTER 11

Adding Context-Sensitive Help

So far, thanks to the Microsoft Foundation Class Library (MFC), Scribble implements a number of common user-interface features, such as print preview and splitter windows. This chapter adds another such feature to Scribble: context-sensitive Windows Help.

Scribble already offers the user some help in the form of prompt strings and tool tips. When the user navigates through a menu by using the UP ARROW and DOWN ARROW keys, or uses the mouse to point to a toolbar button, Scribble displays a brief description of the command's purpose in the status bar (if the status bar is visible); and if the user holds the mouse cursor over a toolbar button, a small pop-up window (called a "tool tip") appears with a brief description of the button.

Since prompts are attached to command IDs, Scribble's toolbar buttons, which duplicate commands on the menus, automatically invoke the appropriate prompts. To get more information on adding tool tips to your application, see the article "Toolbars: Tool Tips" in *Programming with MFC.*

The framework supplies this level of information for commands predefined by the class library. And, as you did in Chapter 6, in "Add the Clear All Command to Scribble's Edit Menu," you can add prompt strings and tool tips to the menu items you create by filling in a field in the menu's property page.

The level of help described in this chapter, however, goes much further. By the end of this chapter, which covers Step 6 of Scribble, Scribble provides access to Windows Help from the Help menu, and to context-sensitive Help by pressing the F1 key or SHIFT+F1. When the user chooses the Help menu item, the Help file for Scribble opens, displaying the Help contents screen. Context-sensitive help invokes a Help topic specific to the area of the user interface with focus when F1 or SHIFT+F1 is pressed.

What This Chapter Covers

This chapter explains how to implement:

- F1 help
- SHIFT+F1 help mode
- Help menu support

The next section, "What Does Context-Sensitive Help Consist of?," describes the three kinds of help listed here.

AppWizard provides this level of help support for free when you choose the Context-Sensitive Help option when initially creating your application starter files. For a quick preview of the results, follow the instructions described in See "Context-Sensitive Help in Action."

Similarly to Chapter 9, "Enhancing Views," this chapter shows how to add functionality supported by AppWizard to your application, if you didn't originally choose it in AppWizard. In "Adding Help to Scribble After the Fact," you'll learn how to add Help support to Scribble, since you didn't select the help option in AppWizard when you originally created your Scribble starter files. This will demonstrate in detail all the work AppWizard does for you to provide a robust Help system for your application.

For an overview of the framework's help support, see Chapter 4, "Working with Dialog Boxes, Controls, and Control Bars" in *Programming with MFC.*

Completing Scribble Step 6

This chapter covers Step 6 of Scribble. If you are working along, begin with the files in your Scribble project directory.

After following the steps descibed in this chapter, your files should closely resemble the files in the SCRIBBLE\STEP6 subdirectory.

If, on the other hand, you want to read along without adding code, refer to the files in the SCRIBBLE\STEP6 subdirectory.

Note If you have not made a local copy of the sample source code for this tutorial step and you wish to do so, see "Installing the Sample Files."

You can also easily preview a running version of Scribble as it appears at the completion of this tutorial step. For more information, see "Previewing the Sample Applications."

What Does Context-Sensitive Help Consist of?

This section provides more detail about the different levels of application help supported by the framework.

- F1 help

  This level of help support enables the user to press the F1 key from an active window, dialog box, or message box, or with a menu item or toolbar button selected, to invoke a Help topic relevant to the selected item.

  For menu items, the user can use the arrow keys to highlight a particular menu item, and then press F1. For toolbar buttons, the user can use the mouse to hold down the button and press F1 before letting the button up.

  If no user interface object is selected, or if no specific Help topic exists, pressing F1 invokes the main Contents topic for the application Help file.

  Note You can define a key other than F1 for help, but it is common among applications for Windows to use F1.

- SHIFT+F1 help mode

  This level of help support enables the user, from within an active application, to press SHIFT+F1 to put the application into "help mode."

  While the application is in help mode, the cursor changes to an arrow beside a question mark. So long as help mode is active, the user can click any window, dialog box, message box, menu item, or toolbar button to summon help specific to the item. This invokes the application's Help file, and ends help mode. Pressing ESC or switching away from the application and back also ends help mode.

  Note When you choose the context-sensitive Help option from AppWizard, the toolbar resource that AppWizard creates includes an additional button that the user can use to invoke help mode. The graphic on the button resembles the mouse cursor as it appears in help mode.

In addition to context-sensitive Help, most applications provide help support through one or more menu items. For instance, most Windows applications include a Help menu item that invokes the application's Help file when chosen. Additional items on the Help menu might, for example, display a Search dialog.

AppWizard Help Support

You can use AppWizard to generate a starter application with all the application help support described here, and more. The following section," Implementing Context-Sensitive Help with AppWizard," describes how to create this starter application. The section "Help Support AppWizard Provides" describes the code and files AppWizard generates when you do so.

Implementing Context-Sensitive Help with AppWizard

You can use AppWizard to automatically enable the framework's support for context-sensitive Help and the Help menu. The following sections explain how to select this option in AppWizard, and what AppWizard creates as a result.

You'll see how to add help support to Scribble later in this chapter, in "Adding Help to Scribble After the Fact."

Note The following procedure is the first step to implementing help support in Scribble.

The Context-Sensitive Help Option

This procedure describes how to select the Context-Sensitive Help option when you create a new application with AppWizard.

▶ To generate AppWizard application help support

1. From the File menu, choose New.
2. In the New dialog, select Project Workspace.

   The New Project Workspace dialog box appears.
3. In the Name box, specify a project name.

   If you are performing this step as the first part of implementing help support in Scribble, give this project a name of HelpApp. We'll refer to this project name throughout the steps in "Adding Help to Scribble After the Fact."
4. Use the Location box to specify the directory where you want to build this sample application.
5. Click Create

   AppWizard creates the project directory, and the MFC AppWizard–Step 1 dialog box appears.
6. Choose the Next button to accept the default options in the AppWizard Step 1, 2 and 3 dialog boxes.
7. In the Step 4 dialog box, select the Context Sensitive Help option, and then click Finish.

   The New Project Information dialog box appears.
8. Click OK to create your application.

You can freely use the Help files that AppWizard creates in your applications and freely ship the compiled help.

Note By default, AppWizard generates the Help Project file in Windows 3.1 format. You can easily upgrade to the Windows 4.0 Help format. For more information, see "Upgrading Your Help Project File to Windows 95," later in this chapter.

Help Support Provided by AppWizard

In addition to the other features AppWizard provides in the skeleton application files, AppWizard adds the following items to support context-sensitive Help:

- Message-map entries in your derived frame-window class (`CMainFrame`) for handler functions to handle Help menu items and F1 and SHIFT+F1 help. These handlers are predefined by the framework.

  To view the message map entries, look in the message map of file MainFrm.cpp, under the \\global help commands section.

- A new Help Topics item in the menu definitions, including a status bar prompt.

  To view the menu item, open the project resource file and view the project-specific menu resource (for example, `IDR_HELPAPTYPE`); to view the status bar prompt string, choose Properties from the Edit menu, and examine the Prompt edit box in the Menu Item Properties page.

- A Windows Help Project file with an .HPJ extension, named for your project.

  This is located in the project \HLP directory. You edit this file to reflect the changes you make to your application Help system.

  Note Help Project files and Windows Help tools are explained in *Programming Tools for the Microsoft Windows Operating System.*

- One or more RTF-format files (.RTF extension) containing help contexts, and boilerplate text.

  These files are located in the project \HLP directory. You can add application-specific help contexts and text to these files to customize your application Help system, and add new topics or delete unused topics. For more information, see the article "Help: Authoring Help Topics" in *Programming with MFC.*

  You can fine-tune your application help further by overriding portions of the class library to support more specific help contexts, such as individual controls in a dialog box. For more information about fine-tuning context-sensitive Help, see Technical Note 28 under MFC Technical Notes in Books Online.

- Several bitmap (.BMP) files used to display graphics in the Help system.

  These are located in the project \HLP directory.

- A .CNT file which contains the information needed to create the help Contents screen.

- A batch file called MAKEHELP.BAT that you can use to compile your Help Project files.

  This file is located in the root project directory.

  Note When you first build the skeleton AppWizard application with context-sensitive Help support, Visual C++ compiles the Help files for you, so that you can immediately take advantage of this feature. Once you customize the Help files, you must recompile them to incorporate your changes into the application project. For more information, see "Compiling Your Help Files."

The next section, See "Context-Sensitive Help in Action," makes a few suggestions for ways you might explore the free application help support that AppWizard implements for you.

See Context-Sensitive Help in Action

Once you've created an AppWizard application with the Context-Sensitive Help option, it's easy to try out the help support provided by the framework and AppWizard.

▶ To try out the help support

- Build the HelpApp application you created in the procedure "To generate AppWizard application help support."

  The first time you build this application, Visual C++ automatically compiles the AppWizard-generated Help Project files as part of the project build process. You don't need to compile them separately until you begin modifying them.

Once the project has been built, you can run the application and try out the various help options. Here are some suggestions for what to try:

- Choose Help Topics from the Help menu.

  This invokes the application Help file generated for you by AppWizard, displaying the Contents tab of the Help Topics dialog. The Contents tab already contains two top-level topics: Menus, which expands to display the menu topics AppWizard generated for the application; and a place-holder topic where you can add your own application-specific topics.

- Expand the top-level Menus topic.

  You'll see all the topics AppWizard provided for you. They describe the standard menus that the framework provides. Continue choosing topics for a particular menu, such as the File menu, until you see a topic for a specific menu item or dialog, such as the Print Preview command or the Print Setup dialog box. You'll see that the Help file is already quite robust.

- Click the Index button.

  Select an index entry, and then choose Display. Again you'll see that the skeleton Help file already contains much relevant information, implemented in a useful help structure.

- Click the Contents button and then select the Find tab.

  Follow the simple instructions for the Find Setup Wizard, which sets up the full-text search capabilities of the Windows help engine. Once finished, your simple Help system suddenly becomes more sophisticated -- the user can type in words or phrases, and if there is a match in any of the AppWizard-generated Help topics, that topic appears in the topic list. Choose Display to see the topic.

  When you add new files to your Help system, recompile, and run the Find Setup Wizard again, the Windows Help engine recreates this full-text search index, incorporating the text in your new topics seamlessly.

- Click the help-mode button on the HelpApp toolbar, which appears as an arrow beside a question mark.

  To get help for a menu item, drop down a menu and click a menu item with the mouse. Click the help-mode button again and then click another toolbar button. Finally, enter help mode again by pressing the SHIFT+F1 keys; then click the toolbar itself, or a window's title bar, or some other element of HelpApp's user interface.

- Drop down a menu and select a menu item by using the DOWN ARROW key. Then press the F1 key to get help for the selected item.

- Press and hold down a toolbar button, and press F1 before releasing the toolbar button.

Thanks to AppWizard and the framework, you—and your users—get all of this help essentially for free.

Compiling Your Help Files

AppWizard provides you with a robust help package: .RTF files that already document the standard (AppWizard-generated) menus, a help project (.HPJ) file, a Help contents topic (.CNT) file, and the other resources needed to generate an application with a fully functional online Help system.

You'll undoubtedly want to take the files that AppWizard provides and customize them to your particular application, even as you develop the application. When you compile and build your developing application, you also need to recompile your project Help files so that they continue to reflect your customizations.

Note The article Help: Authoring Help Topics in *Programming with MFC* explains how to author the Help topics using a program that can edit .RTF files, such as Microsoft Word for Windows.

There are two ways you can compile the Help files you create for your Visual C++ applications:

- Compile the help project (.HPJ) file from within the Microsoft Developer Studio.

  This calls MAKEHELP.BAT, a batch file which builds the Help system.

- Run the MAKEHELP.BAT file from the command line.

  MAKEHELP.BAT is generated by AppWizard each time you create the starter files for an application that contains AppWizard-provided context-sensitive Help support.

Note By default AppWizard generates the Help Project files in Windows 3.1 format. When you use the VC++ tools to compile your Help project, the result is a system compatible with either Windows 3.1 or 4.0. If porting your applications to other environments is not an issue, you can also easily convert your Help project to the 4.0 format. For more information, see "Upgrading Your Help Project File to Windows 95," later in this chapter.

Compiling Your Help Files from within Microsoft Developer Studio

For this option to work, the executable files called by MAKEHELP.BAT must be in a directory listed in the Directories tab of the Options dialog. (Choose Options from the Tools menu.) By default, these files are installed to the \BIN directory of your Visual C++ installation, and included in the Options dialog, so unless you modify either of these conditions your Help compilation will be a completely transparent part of building your project files.

For more infomation about directory settings, see "Setting Directories" in Chapter 23 of the *Visual C++ User's Guide*.

▶ To compile your Help files from within Microsoft Developer Studio

1. From the Build pane of the Project workspace, select your help project (.HPJ) file. (You needn't open the file; but it must remain selected while you perform the next step.)
2. From the Build menu, choose Compile.

   This calls MAKEHELP.BAT, which in turn calls the following two programs:

   - MAKEHM.EXE, a program that incorporates your context-sensitive topic IDs
   - The Windows Help Compiler, which builds your .HLP file.

Compiling Your Help Files from the Command Line

In order to compile your application Help from the command line, the Windows Help Compiler and MAKEHM.EXE must be in your path statement. To ensure that this is the case (and to set other relevant environment variables for a specific build target) you may want to run the VCVARS32 batch file. In general, it's a good idea to run

VCVARS32 prior to running build tools such as MAKEHELP from the command line.

▶ To run VCVARS32.BAT

- At the command line, type:

  VCVARS32 [*target*]

  where target is one of the following: x86, m68k, mppc.

▶ To compile your Help files from the command line

1 At the command line, change to the root directory of your help project.

  AppWizard copies MAKEHELP.BAT to this directory by default.

2 Type MAKEHELP and press ENTER.

  The Windows Help compiler generates your application help (.HLP) file.

Upgrading Your Help Project File to Windows 95

Because Visual C++ version 4.0 ships with the Windows 4.0 Help compiler, you can easily upgrade your Help Project files to a 4.0 format. This gives you access to the Windows 4.0 Help Workshop, a graphical Help authoring environment with many useful features.

Note If you're planning on porting your applications to other platforms, first ensure that those platforms have a compiler that's compatible with the 4.0 Help Project file before upgrading.

▶ To upgrade your project file

1 Start the Help Workshop by running HCW.EXE.

  This file is installed to the \BIN directory of a default product installation.

2 In Help Workshop, open your HPJ file.

  You'll notice that Help Workshop automatically adds entries to the file, such as the LCID (language) entry in the [OPTIONS] section.

3 Use Help Workshop to make any modifications, and then save your file.

  Help Workshop saves your file in the 4.0 format.

For more information about Help Workshop, see the online Help file included with the product. This file, HCW.HLP, is installed to the \HELP directory of a default Visual C++ installation. You can open this file directly, by double-clicking it, or view it by opening Help Workshop and then choosing Help.

Adding Help to Scribble After the Fact

This section explains how to add context-sensitive Help to an application when you did not originally choose it as an AppWizard option.

Merging context-sensitive Help support into Scribble at a later stage requires several general steps. Each step is explained in more detail in following sections. The overall steps are:

1. Run AppWizard and choose the context-sensitive Help option to create a new application that has the help-related files and code.

   See the procedure "To generate AppWizard application help support." You'll use this skeleton application to copy code and resources into Scribble.

2. Copy resources from this application to Scribble.

   See the section "Copying Help-Specific Resources to Scribble."

3. Copy help-related files and code from the application to Scribble.

   See the section "Copying the Help-Related Code and Files to Scribble."

4. Customize the contents of the files you copied from HELPAPP.

   See the section "Customizing the Help Files and Code for Scribble."

5. Build the new version of Scribble and compile its Help file.

   See Complete Scribble's Help Implementation.

Copying Help-Specific Resources to Scribble

In the following procedures, you'll copy menu items, accelerator keys, and status-bar prompt strings from HelpApp, (the project you created with AppWizard help support), to Scribble. As you do this, you'll not only learn about adding help to an application after the fact, you'll also learn how easy it is to copy resources from one resource file in Visual C++ to another. Finally, you'll add a toolbar button to Scribble's toolbar resource, just as you did in Chapter 6, "Constructing the User Interface."

Note If you have not already created HelpApp, refer to the procedure "To generate AppWizard application help support." This is the first step in implementing help support in Scribble.

In the following procedures, you'll use the menu editor, the accelerator editor, the string editor, and the Toolbar editor, respectively. For more information about these resource editors, see Chapters 7, 8, 9 and 11 of the *Visual C++ User's Guide*.

Copying the Help Menu Resources

First, copy Help-related menu resources from HelpApp to Scribble.

▶ To copy Help menu resources to Scribble

1 Open SCRIBBLE.RC, and HELPAPP.RC.

2 Open the **IDR_MAINFRAME** menus from both resource files.

Arrange the menu editor windows so that they don't overlap.

3 Drop down both Help menus.

4 Click the Help Topics menu item in the HelpApp Help menu. Then hold down the SHIFT key while you click on the separator below the Help Topics item. Release the SHIFT key.

This selects the menu item and separator.

5 Hold down the CTRL key, and use the mouse to drag the highlighted menu items to the Help menu in SCRIBBLE.RC, above the About Scribble menu item. Release the mouse button and the CTRL key.

This copies the menu item and the separator to Scribble.

6 Close the two **IDR_MAINFRAME** menu editing windows.

7 Repeat steps 3 through 6, using the application-specific menu resources in Scribble and HelpApp: `IDR_SCRIBBTYPE` and `IDR_HELPAPTYPE`, respectively.

Copying the Help Accelerator Resources

Next, copy the accelerator resources for the menu items.

▶ To copy the accelerator keys for Help resources

1 Open the Accelerator table resource, **IDR_MAINFRAME**, from both Scribble and HelpApp, again resizing the Accelerator editor windows so that they do not overlap.

2 Select the **ID_HELP** command from the HelpApp Accelerator table.

3 Hold down the CTRL key while you drag this entry to the Scribble Accelerator editor window.

It doesn't matter where in the window you drop the entry. This copies the accelerator keys (F1 and SHIFT+F1) for the Help menu item to Scribble.

4 Repeat this procedure for the **ID_CONTEXT_HELP** command.

This maps to the help mode toolbar button, described below in Copying the Help Mode Toolbar Button.

5 Close both Accelerator editor windows.

Copying the Help-Related String Resources

In this procedure you copy the command IDs for Help-related menu items.

▶ To edit Help-related string resources

1 Open the String Table resource for both Scribble and HelpApp, resizing both editor windows so they do not overlap.

2 Locate the **AFX_IDS_IDLEMESSAGE** string in both tables, noting the difference in the captions.

For an application without help, AppWizard defines the default status-bar prompt to be "Ready". This is the string displayed in the status bar when no other command prompt is being displayed. This string resource is identified as **AFX_IDS_IDLEMESSAGE**.

3 Select **AFX_IDS_IDLEMESSAGE** in Scribble's string table, and choose Properties from the Edit menu to open the String Properties window.

4 Edit the Caption so that it matches that in the HelpApp version of this string resource: For Help, press F1.

5 Next, copy the following strings from the HelpApp String Table to Scribble's String Table: **AFX_IDS_HELPMODEMESSAGE**, **ID_CONTEXT_HELP**, and **ID_HELP**.

The copying procedure is similar to the procedures for copying menus and accelerators. To copy several contiguous strings, hold the SHIFT key down while selecting the strings. Then hold the CTRL key down while dragging the selected strings to the new window.

The **AFX_IDS_HELPMODEMESSAGE** prompt displays in the status bar while the application is in help mode. The **ID_CONTEXT_HELP** prompt displays when the mouse pauses over the help mode toolbar button.

Notice that Scribble's string table already contains the **ID_HELP_FINDER** string, representing the prompt that displays for the Help Topics command on the Help menu. This was copied when you copied the Help Topic menu item.

The **ID_HELP** command is called when the user presses F1.

6 Close the String Table resources.

Copying the Help Mode Toolbar Button

When you generate context-sensitive Help from AppWizard, AppWizard creates a button on the toolbar which, when chosen, places the application in Help mode, just as pressing SHIFT+F1 does.

In this procedure you'll copy the Help mode button to Scribble's toolbar resource from HelpApp's toolbar resource, simply by dragging it.

▶ **To copy the help-mode toolbar button**

1 Open Scribble's toolbar resource, and HelpApp's toolbar resource, resizing both editor windows so they do not overlap. and so you can see both subject toolbars.

2 Press CTRL and drag the Help mode button from HelpApp's toolbar onto Scribble's toolbar.

3 Save the Scribble resource file, Scribble.rc, and close the HelpApp resource file, Helpapp.rc.

Copying the Help-Related Code and Files to Scribble

AppWizard generates not only help-specific code, but files that are used to build the application's Help system. In the following procedures, you'll copy those files, and the Help-related code from HelpApp into the Scribble project.

Copying the Help Message Map Commands

AppWizard places Help-specific code into the message map in the frame window class implementation file, MainFrm.cpp.

▶ **To copy help-related code to Scribble**

1 Open the HelpApp project MainFrm.cpp file, and your Scribble project version of MainFrm.cpp.

2 Scroll to the Help-related code, delineated by the `//global help commands` comment, in the message map section of the HelpApp MainFrm.cpp file.

3 Copy this code, shown below, from the HelpApp MainFrm.cpp file, and paste into the same position in the message map in Scribble's MainFrm.cpp.

```
// Global help commands
ON_COMMAND(ID_HELP_FINDER, CMDIFrameWnd::OnHelpFinder)
ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpFinder)
```

You've already seen how **ID_HELP_FINDER** and **ID_CONTEXT_HELP** are called. When the user presses F1, the framework calls **ID_HELP** and directly handles this command so long as the application contains help support.

4 Close the HelpApp project's MainFrm.cpp file, and save the changes to Scribble's MainFrm.cpp.

Copying the Help-Related Files to Scribble

To perform the following procedure, use File Manager or another file management utility.

▶ To copy help-related files to Scribble

1 Copy the MAKEHELP.BAT file from the HelpApp project (root) directory to your Scribble project (root) directory.

2 Select and copy the entire HELPAPP\HLP directory to the Scribble project directory, thereby creating the \HLP subdirectory and all its files under the Scribble project.

The files contained in the \HLP directory include, among others:

- The Help Project (.HPJ) file, which tells the Help compiler how to build the Help file.
- A file with the extension .CNT, needed to display the Help Contents screen.
- A file with the extension .HM, created during the process of compiling the Help file, needed to provide context-sensitive Help for user interface objects. (If you did not build the HelpApp project, this file won't have been created yet.)
- Several bitmap (.BMP) files used to display graphics in the Help system.

3 In the Scribble project \HLP directory, rename HELPAPP.HPJ to SCRIBBLE.HPJ; HELPAPP.HM to SCRIBBLE.HM (if this file exists); and HELPAPP.CNT to SCRIBBLE.CNT.

The bitmap and .RTF files do not need to be renamed.

4 Copy the following file from the Scribble Step 6 source code directory to your corresponding Scribble directory:

- PEN.RTF (in the \HLP directory)

  This file contains the Scribble-specific Help Contents topic, and Help topics for Scribble's Pen menu.

Note If you have not made a local copy of the sample source code for Scribble Step 6, see "Installing the Sample Files."

The next section, "Customizing the Help Files and Code for Scribble," describes how to modify the files you just copied so they work with your Scribble project.

Customizing the Help Files and Code for Scribble

Once the Help-specific files are copied and renamed, you need to modify their contents to refer to the Scribble project instead of to HelpApp. You can perform the following procedure in the Visual C++ editor window.

▶ To customize the Help files for Scribble

1 Use the File Open command to open the Scribble copy of MAKEHELP.BAT, and change all occurrences of the string "HelpApp" to "Scribble."

2 Save and close MAKEHELP.BAT.

3 Repeat step 1 for the files SCRIBBLE.CNT and SCRIBBLE.HPJ (located in the \HLP directory).

Keep both files open.

4 In SCRIBBLE.CNT, Add an entry in the menu structure for the Pen menu, between the Edit and View entries:

```
2 Pen menu=menu_pen
```

5 Save and close SCRIBBLE.CNT.

6 In SCRIBBLE.HPJ, make the following changes:

- Under the [FILES] section, add the line

  ```
  pen.rtf
  ```

  This adds the file, PEN.RTF, to your Help project.

- Under the [OPTIONS] section, change "CONTENTS=main_index" to "CONTENTS=new_index".

  `Main_index` is the context string for the Help Contents topic that AppWizard created; replacing it with `new_index` calls the topic, in PEN.RTF, for Scribble's Help Contents screen instead.

- Under the [ALIAS] section, change the string "HIDR_MAINFRAME = main_index" to "HIDR_MAINFRAME = new_index"; and change "HIDR_HELPAPTYPE" to "HIDR_SCRIBBTYPE".

The next topic, Scribble's Help Project File, shows the edited .HPJ file.

Scribble's Help Project File

Here's how the edited Scribble.hpj should look:

```
[OPTIONS]
CONTENTS=new_index (or Main_index)
TITLE=Scribble Application Help
COMPRESS=true
WARNING=2
BMROOT=..,.
ROOT=..,.

[FILES]
afxcore.rtf
afxprint.rtf
pen.rtf

[ALIAS]
HIDR_MAINFRAME = new_index (or Main_index)
HIDR_SCRIBBTYPE = HIDR_DOC1TYPE
HIDD_ABOUTBOX = HID_APP_ABOUT
HID_HT_SIZE = HID_SC_SIZE
HID_HT_HSCROLL = scrollbars
HID_HT_VSCROLL = scrollbars
HID_HT_MINBUTTON = HID_SC_MINIMIZE
HID_HT_MAXBUTTON = HID_SC_MAXIMIZE
```

```
AFX_HIDP_INVALID_FILENAME          = AFX_HIDP_default
AFX_HIDP_FAILED_TO_OPEN_DOC        = AFX_HIDP_default
AFX_HIDP_FAILED_TO_SAVE_DOC        = AFX_HIDP_default
AFX_HIDP_ASK_TO_SAVE               = AFX_HIDP_default
AFX_HIDP_FAILED_TO_CREATE_DOC      = AFX_HIDP_default
AFX_HIDP_FILE_TOO_LARGE            = AFX_HIDP_default
AFX_HIDP_FAILED_TO_START_PRINT     = AFX_HIDP_default
AFX_HIDP_FAILED_TO_LAUNCH_HELP     = AFX_HIDP_default
AFX_HIDP_INTERNAL_FAILURE          = AFX_HIDP_default
AFX_HIDP_COMMAND_FAILURE           = AFX_HIDP_default
AFX_HIDP_PARSE_INT                 = AFX_HIDP_default
AFX_HIDP_PARSE_REAL                = AFX_HIDP_default
AFX_HIDP_PARSE_INT_RANGE           = AFX_HIDP_default
AFX_HIDP_PARSE_REAL_RANGE          = AFX_HIDP_default
AFX_HIDP_PARSE_STRING_SIZE         = AFX_HIDP_default
AFX_HIDP_FAILED_INVALID_FORMAT     = AFX_HIDP_default
AFX_HIDP_FAILED_INVALID_PATH       = AFX_HIDP_default
AFX_HIDP_FAILED_DISK_FULL          = AFX_HIDP_default
AFX_HIDP_FAILED_ACCESS_READ        = AFX_HIDP_default
AFX_HIDP_FAILED_ACCESS_WRITE       = AFX_HIDP_default
AFX_HIDP_FAILED_IO_ERROR_READ      = AFX_HIDP_default
AFX_HIDP_FAILED_IO_ERROR_WRITE     = AFX_HIDP_default
AFX_HIDP_STATIC_OBJECT             = AFX_HIDP_default
AFX_HIDP_FAILED_TO_CONNECT         = AFX_HIDP_default
AFX_HIDP_SERVER_BUSY               = AFX_HIDP_default
AFX_HIDP_BAD_VERB                  = AFX_HIDP_default
AFX_HIDP_FAILED_MEMORY_ALLOC       = AFX_HIDP_default
AFX_HIDP_FAILED_TO_NOTIFY          = AFX_HIDP_default
AFX_HIDP_FAILED_TO_LAUNCH          = AFX_HIDP_default
AFX_HIDP_ASK_TO_UPDATE             = AFX_HIDP_default
AFX_HIDP_FAILED_TO_UPDATE          = AFX_HIDP_default
AFX_HIDP_FAILED_TO_REGISTER        = AFX_HIDP_default
AFX_HIDP_FAILED_TO_AUTO_REGISTER   = AFX_HIDP_default

[MAP]
#include <D:\MSDEV\MFC\include\afxhelp.hm>
#include <Scribble.hm>
```

Note If your product installation is in another directory, change the entry in the [MAP] section accordingly.

The next topic, "Completing Scribble's Help Implementation," describes how to build the Scribble Help system.

Completing Scribble's Help Implementation

The first time you build an AppWizard-generated project with Help support, Visual C++ builds your Help project files for you. Thereafter, if you change the Help files, you must recompile them just as you would any other project file. As described in

"Compiling Your Help Files," there are two ways to do this; you can either run a batch file from the command prompt, or compile the Help Project file from within Microsoft Developer Studio.

To compile Scribble's Help Project file from within Microsoft Developer Studio, you must first perform two steps:

- Add Scribble.hpj to the Scribble project
- Specify Custom Build settings

Both of these steps are optional: you can compile Scribble's Help files from the command line without any further customizations. (Note that AppWizard performs these steps for you automatically when you choose the context-sensitive Help option.)

Compiling Scribble's Help from Microsoft Developer Studio

You can easily prepare Scribble to compile the new Help files from within Microsoft Developer Studio, by performing the following two procedures.

▶ To add Scribble.hpj to the project

1. With your Scribble project open in Microsoft Developer Studio, from the Insert menu, choose Files to Project.
2. In the Insert Files to Project dialog, navigate to the \HLP directory underneath the Scribble project root directory, and specify Scribble.hpj.
3. Choose Add.

   The file now appears in the FileView pane of the workspace.

When AppWizard adds context-sensitive Help support, it implements special build options for the application's Help Project file. You can duplicate this functionality by specifying Custom Build settings for Scribble.hpj. Once you complete this step, you can build Scribble.hpj from within Microsoft Developer Studio.

▶ To specify custom build settings

1. From the Build menu, choose Settings.

   The Project Settings Dialog opens.
2. In the Settings For area, expand the Scribble project folder and select Scribble.hpj.

   You can simultaneously select Scribble.hpj in both the Debug and Release folders if you want to implement this feature for both project build types.
3. Select the Custom Build tab.
4. In the Build Command(s) edit box, type the following:

   ```
   $(ProjDir)\makehelp.bat
   ```

   This calls MAKEHELP.BAT, which in turn compiles the Help files.

5 In the Output File(s) edit box, type the following:

```
.\$(OutDir)\$(TargetName).hlp
```

This specifies Scribble.hlp as the output file.

6 Click OK.

You can now compile Scribble.hlp from Microsoft Developer Studio anytime you need to update the contents of the Help Project files.

▶ To compile Scribble Help from within Microsoft Developer Studio

1 After completing the previous two procedures, in FileView, select Scribble.hpj.

2 From the Build menu, choose Compile Scribble.hpj.

You can also simply build Scribble.exe at this point to see Scribble running with full application Help support.

▶ To try out Scribble's Help

1 From the Help menu, choose Help Topics.

2 In the Contents tab, expand the Menus heading, and double-click the Pen menu topic.

This is the topic you added to Scribble's Help system by editing the .CNT and .HPJ files. You'll see the custom Help that has been provided in PEN.RTF.

3 Return to the Scribble application, and press SHIFT+F1 to enter Help mode, then click one of the items on Scribble's Pen menu.

Help mode, invoked by pressing SHIFT+F1, is another way for users to access Scribble application Help topics.

4 Select the Index and Find commands in the Help topics dialog box to see the new Windows 4.0 application Help interface.

Compiling Scribble Help from the Command Line

To successfully run MAKEHELP.BAT from the command line, two files must be in your path statement: the Windows Help Compiler (HCW.EXE), and MAKEHM.EXE, a file which incorporates your context-sensitive topic IDs into the built Help system. These files are copied to your installation \BIN directory as part of a standard setup.

▶ To compile Scribble's Help from the command line

1 From the command line, switch to the Scribble project root directory.

2 Run VCVARS32.BAT.

This sets up your environment variables.

3 Run MAKEHELP.BAT.

4 In Microsoft Developer Studio, rebuild Scribble.

For more information on how to create a Help file for an MFC application, see the "Help" articles in *Programming with MFC*.

This concludes Step 6. Chapter 12 describes Step 7, which adds OLE server support to Scribble so that the user can embed Scribble objects into OLE container documents.

CHAPTER 12

Creating an OLE Server

An OLE in-place editing server application can create OLE items that can be embedded or linked into container applications. Some server applications support only the creation of embedded items, while others support the creation of both embedded and linked items.

A container application must be able to start a server application when the user wants to edit an item. If a server application supports linked items, it must also be able to copy its data to the Clipboard so that a container can use that data to create OLE items.

An application can be both a container and a server; that is, it can both incorporate external data into its documents and create data that can be incorporated as items into the documents of other applications. For more information on OLE containers and servers, see the article "OLE Overview: Containers and Servers" in *Programming with MFC.*

Scribble Step 7 addresses two general cases for adding OLE server support to an application:

- Creating a new OLE in-place editing server application from scratch
- Adding OLE in-place editing server support to an existing application

The technique described in this chapter illustrates both cases, even though the tutorial starts with an existing MFC application, Scribble Step 6, to which you will add OLE server support. For more information on Scribble, see Chapters 2 through 11.

As when adding support for context-sensitive help to Scribble Step 6, you will use AppWizard to provide a skeleton OLE server application in a scratch directory. Then you will copy files and code fragments from the scratch directory to the existing Scribble code base. By doing this, you will learn about the OLE server code that AppWizard provides. Therefore, even if you are starting a new MFC OLE server application from scratch, you are advised to read this tutorial, if not actually do the steps.

How does this approach differ from the traditional approach of copying source code from a sample application? AppWizard allows you to customize the sample code you will be borrowing. That is, when you create the scratch application, you will name it "Scribble," give the classes the same names Scribble itself uses, and so on. Thus, when you copy source code from the AppWizard-created sample application, it will match the class names of your original application. You can use this approach to add other AppWizard-supported features to your existing MFC applications "after the fact."

Previewing Scribble Running as an OLE Server

Before working through the steps of adding OLE server support to Scribble, try out the completed application. This will help you appreciate how Scribble behaves when it is activated by an OLE container.

The OLE container application you will use is the completed Container tutorial project (you'll build it from the source files). Before you can view Scribble (the server application) files from within Container, you need to register Scribble as an OLE server application.

▶ To install and register Scribble as an OLE server application

- Build and run Scribble.exe from the sample source files for Step 7, or run Scribble.exe directly from the Books Online.

  Running Scribble briefly as a stand-alone application registers it in the system registry as an OLE server.

  Note To install the Sample files, see "Installing the Sample Files." To run an application from Books Online, see "Previewing the Sample Applications."

▶ To preview the Container project as an OLE container application

1. Build and run Container.exe from the sample source files for Step 2, or run Container.exe directly from the Books Online.
2. From Container's Edit menu, choose Insert New Object.

   The Insert Object dialog box appears.
3. In the Object Type list, select Scribb Document.

   This opens a Scribble document inside the Container application. Notice how Scribble:

   - Opens a window inside Container for in-place editing. The window has a resizeable border so you can change the size of the window while visually editing the Scribble object.
   - Takes over part of Container's menu bar and adds its own Pen menu.

- Takes over Container's toolbar.

4 Draw something in Scribble's in-place window.

Notice how drawing in Scribble's in-place window is just like drawing in the stand-alone Scribble application.

5 Click outside the window.

With focus taken away so the Scribble document is no longer selected, the Scribble object is redrawn inside Container's view with the help of the Scribble server.

The rest of this chapter describes how to make Scribble an OLE server.

Steps to Provide OLE Server Support After the Fact

When you first used AppWizard to create your Scribble starter files, you could have chosen the Full Server option to create Scribble as an OLE server application. However, the purpose of this tutorial step is to show you how to build OLE server support into your existing MFC application.

You'll complete the following steps:

- Run AppWizard with the full server option.
- Copy new files from the AppWizard full server project to your Scribble project directory.
- Add AFXOLE.H to your precompiled header file.
- Add OLE-specific server code to the application object.
- Convert the **CDocument** class to the **COleServerDoc** class.

In addition, you'll take several steps to:

- Edit OLE-related resources.
- Add application-specific server support.

Using AppWizard's Full Server Option

This step of Scribble is similar to Step 6 ("Adding Context-Sensitive Help") in that you'll start by running AppWizard with an option that you didn't originally choose. In this case, the option is OLE Server support. Then you'll copy the files and resources that AppWizard generates for an OLE Server application to your Scribble project.

▶ To set the AppWizard options for an OLE server application

1 From the File menu, choose New.

The New dialog box appears.

2 Select Project Workspace, and click OK.

The New Project Workspace dialog box appears.

3 In the Project Name box, type `Scribble`.

You're using the same project name (Scribble) so that you can copy code that contains class names and other identifiers into your current Scribble project files.

However, you'll need to make sure you create this Scribble in a different project directory than the original Scribble.

4 Use the Browse button next to the Location box to navigate to a directory of your choice.

5 In the Location box, type `Scratch` as the new project directory.

This creates the new Scribble project under a subdirectory named Scratch.

6 In the Type listbox, make sure MFC AppWizard (exe) is specified.

7 If any check boxes other than Win32 appear in the Platforms box, clear them.

Note OLE support, as required by this step of the tutorial, will not be generated if any other check boxes are selected.

8 Click Create.

9 Respond Yes when prompted about creating the Scratch directory.

AppWizard creates the project directory, and the MFC AppWizard–Step 1 dialog box appears.

10 Click Next in the dialog boxes for AppWizard Steps 1 and 2 to accept the default options.

11 In the AppWizard–Step 3 dialog box, select Full-Server, and then click Next.

12 In the AppWizard–Step 4 dialog box, click the Advanced button.

13 In the File Extension box, type `scb`.

The entry in the Filter Name box box changes appropriately.

14 Click Close.

15 Choose the Context Sensitive Help option, and click Next.

When you choose Context-Sensitive Help with the Full Server option, AppWizard creates:

- The toolbar bitmap for Scribble when it runs as a server
- An additional .RTF file for the OLE commands that appear on the menu when Scribble is running as a server.

You'll copy these resources to your Scribble project later.

16 In AppWizard–Step 5, click Next to accept the default options.

17 In the `AppWizardStep` 6 dialog box, check and modify class names and filenames to make them match the original Scribble application.

Important If you are starting from the sample source code files for Step 6, rather than from a Scribble project which you have been developing step by step, you need to modify the filenames and classnames differently than what is described here. Use the alternate filenames described in the topic "Using Short Filenames," after this procedure.

For more information about using short filenames, see the topic "A Note About Long Filenames."

- Select the class `CScribbleView`, and choose **CScrollView** in the Base Class box.
- Select the class `CScribbleSrvrItem`, and change its name to `CScribbleItem`, the header filename to **ScribbleItem.h**, and the implementation filename to **ScribbleItem.cpp**.

  Notice that **COleServerItem** is the base class, which reflects your full-server choice.

18 Choose Finish.

19 Choose Yes if the following message box appears:

```
A unique class ID already exists in the registration database for this
document type. Use existing ID?
```

This message box appears when you have already run the OLE server version of Scribble, to register it with Windows.

20 In the New Project Information dialog box, confirm the specifications and click OK.

AppWizard creates the new files.

21 From the File menu, choose Close Workspace to close the SCRATCH\Scribble project.

In the next procedure you'll incorporate files and resources from this SCRATCH version of Scribble, into the Scribble you've been developing all along.

Using Short Filenames

If you are starting Scribble Step 7 from the Step 6 sample source files, your project filenames need to match the short filenames found on the distribution CD. (Class names remain the same.) The Visual C++ distribution CD uses short (8.3) filenames in order to meet ISO standards.

The following list describes the filenames you need to change. For convenience, class name modifications for this step, as described in the previous procedure, are listed as well.

Note As described in the previous procedure, you need to modify these project file and classnames in the AppWizard–Step 6 dialog before creating your project.

- For class `CScribbleSrvrItem`, change the classname to `CScribbleItem`, the header file to ScribItm.h; and the implementation file to ScribItm.cpp.

- For class `CScribbleView`, change the header file to ScribVw.h and the implementation file to ScribVw.cpp. As mentioned in the previous procedure, change the base class of `CScribbleView` to `CScrollView`.
- For class `CScribbleDoc`, change the header file to ScribDoc.h and the implementation file to ScribDoc.cpp.

Note Keep in mind, as you work through the procedures in this tutorial step, that whenever a procedure refers to a long filename you should substitute the equivalent short filename.

Transfer Scratch Files to Your Scribble Project

AppWizard provides several source files for an OLE server application that you can use as-is in Scribble. The following procedures describe how to incorporate source files and code from the SCRATCH version of Scribble, which you created with the AppWizard full-server option, into the version of Scribble you've been developing step by step, and which did not initially include full-server support.

▶ To add the new files to your Scribble project

1 Copy the following files from SCRIBBLE\SCRATCH to your Scribble project root directory:

- IPFRAME.H
- IPFRAME.CPP
- SCRIBBLE.CPP (choose Yes when prompted to overwrite the existing version of this file)
- SCRIBBLEITEM.H
- SCRIBBLEITEM.CPP
- SCRIBBLE.REG (choose Yes if prompted to overwrite the existing version of this file)
- HLP\AFXOLESV.RTF
- RES\ITOOLBAR.BMP

These files provide "In Place" (IP) editing functionality for Scribble. Placing them in your project directory makes dependent files available to the project. When AppWizard generates Scribble.cpp with Full Server support, it adds OLE-specific code to the file.

2 Start Visual C++, if necessary, and from the File menu, choose Open Workspace.

3 Navigate to your original version of Scribble, and open Scribble.mdp.

4 From the Insert menu, choose Files Into Project.

The Insert Files into Project dialog box appears. By default, the dialog points to your current project directory.

5 Select the files IpFrame.cpp and ScribbleItem.cpp, and choose the Add button.

Visual C++ automatically adds any dependent files, in this case, the header files associated with the two implementation files you just added to the project. You can now view the added files in FileView, and their associated classes in ClassView.

6 From FileView, open SCRIBBLE.HPJ.

Because you added the help source file, AFXOLESV.RTF, to the project, you need to modify the help project file.

7 In the [FILES] section of SCRIBBLE.HPJ, type "afxolesv.rtf," and save the file.

Similar to the process for adding files to a project, you can't just copy a help source (.RTF) file; you must add it to the Help project. Modifying the Help Project (.HPJ) file accomplishes this.

Scribble's In Place Editing Files

IpFrame.cpp contains the implementation of Scribble's **COleIPFrameWnd**-derived class. This is the frame window for Scribble when it draws in the container. **COleIPFrameWnd** provides the resize border that you noticed in the preview demonstration. Note that Scribble only uses this **COleIPFrameWnd** object when the Scribble object is in-place activated in the container application. Only then does the server need to provide a window. When the Scribble object is not activated in place, but is just being drawn in the container's window, the OLE server provides a metafile (a list of drawing commands) to the container so it can then play the metafile.

ScribbleItem.cpp contains the implementation of Scribble's **COleServerItem**-derived class. The **COleServerItem** object represents the Scribble document when it is embedded in a container.

Registering an OLE Server Application with Windows

In order to run your application as an OLE server, you need to register it with Windows. AppWizard provides code that does this for you when you simply run the application, as you saw in the section "Preview Scribble Running as an OLE Server." AppWizard also provides a text file, SCRIBBLE.REG, which you can use to register Scribble as an OLE server. You do this by running REGEDIT and merging SCRIBBLE.REG with the existing Windows registry file.

▶ To use REGEDIT to register Scribble as an OLE Server

1 Run REGEDIT. (Note that on Windows NT, both REGEDIT and REGEDT32 are available. You can only merge a .REG file using REGEDIT.)

2 On Windows NT, from the File menu, choose Merge Registration File. On Windows 95, from the Registry menu, choose Import Registry File.

3 Select SCRIBBLE.REG from your Scribble project directory.

4 When notified that the information has been registered, click OK and then close REGEDIT.

If you're going to distribute your application, you should provide this registration as part of the application setup routine.

There are several ways you can do this:

- Spawn REGEDIT, using the /s (silent) option and specifying the .REG file as a command line parameter, for example:

  ```
  regedit /s scribble.reg
  ```

  Naturally, you'd need to create and maintain a .REG file specific to your particular application.
- Use the Windows registry APIs to set up all the registry keys.
- Let the framework programmatically register the application for you, by allowing your application to be run with a special switch (for example, "/Register") that registers the application and then immediately quits, such that the application can be run with this special switch as part of setup. For example:

  ```
  scribble /register
  ```

  Of course, you'd need to modify your application's `InitInstance` code so that it recognizes this special switch. This works well for applications that are small and boot relatively quickly, since you can leverage the automatic registration that MFC provides as part of the framework.

  For more information, see "Framework Support for Registering the Application with Windows."

Add AFXOLE.H to Your Precompiled Header File

The MFC OLE support is kept in a separate extension header file, AFXOLE.H. Because several SCRIBBLE implementation files refer to the MFC OLE classes, it is a good idea to include it in STDAFX.H, the precompiled header for Scribble.

For more information about precompiled headers, see "Precompiled Headers."

▶ To add AFXOLE.H to the precompiled header file

1. From FileView, open STDAFX.H (under the Dependencies folder), and add the following **#include** statement:

   ```
   #include <afxole.h>      // MFC OLE support
   ```

2. Save the file.

Add OLE Server Support to the Application Object

You've already added much OLE server support to the application object by using the Full Server version of Scribble.cpp. In the next two procedures, you'll complete this process by adding a **COleTemplateServer** data member to `CScribbleApp`, and by adding a string resource to Scribble's string table.

▶ To add a COleTemplateServer data member to CScribbleApp

- Use ClassView to jump to the `CScribbleApp` declaration, and add the following lines at the beginning of the // implementation section:

```
COleTemplateServer m_server;
// Server object for document creation
```

You will find the same code in SCRATCH\SCRIBBLE.H, which was provided by AppWizard.

The **COleTemplateServer** object is used to register a server application with OLE. See how `m_server` is used in Scribble's `InitInstance`.

▶ To copy the string resource

1. Switch to ResourceView and open Scribble's String table.
2. Use the File menu to open the .RC file for the SCRATCH\SCRIBBLE project, and open its String table. (Don't insert this resource file.)
3. Arrange the string editor windows so that you can view both String tables and they don't overlap.
4. Press CTRL, and then drag the resource `IDP_OLE_INIT_FAILED` from the Full Server version of Scribble to your Scribble project's String table.

   This AppWizard provided string is used in Scribble's `InitInstance`. The string text is:

   ```
   OLE initialization failed. Make sure that the OLE libraries are
   correct version.
   ```

   You can view this text by selecting the string and then choosing Properties from the Edit menu.
5. Save your Scribble.rc and close both String tables. (You can also close the SCRATCH\SCRIBBLE version of Scribble.rc.)

Convert the CDocument Class to the COleServerDoc Class

The **CDocument** class implements standard document behavior in a stand-alone application. When the application runs as an OLE in-place editing server, however, the document must do extra work on behalf of OLE. The framework implements the bulk of this OLE document support in class **COleServerDoc**. The remaining work you have to do is:

- Change the base class of `CScribbleDoc` from **CDocument** to **COleServerDoc**.
- Implement the document's support for embedded items.

▶ To change the base class of CScribbleDoc

1 Use ClassView to jump to the declaration for class `CScribbleDoc`, and change:

```
class CScribbleDoc : public CDocument
```

to:

```
class CScribbleDoc : public COleServerDoc
```

2 Use FileView to open ScribbleDoc.cpp, and replace all instances of **CDocument** with **COleServerDoc**.

This changes the base class reference of `CScribbleDoc` from **CDocument** to **COleServerDoc**.

The **COleServerItem** object represents the Scribble document when the document is embedded in a container. To create a **COleServerItem** for a given document, AppWizard provides an override of **GetEmbeddedItem** in the **COleServerDoc**-derived class. The return type of **OnGetEmbeddedItem** is a pointer to a **COleServerItem**.

Note A **COleServerItem** object can also represent an OLE link item, but Scribble doesn't illustrate that. For an illustration of a link item, see the sample HIERSVR under MFC Samples in Books Online.

In the following procedure, you'll fill in some of the code that AppWizard would have generated had you originally chosen the Full Server option.

▶ To implement the document's support for embedded items

1 Switch to ScribbleDoc.h in the editor and add the following forward class reference for `CScribbleItem`, after the forward declaration for class `CStroke`:

```
class CScribbleItem;
```

2 In ScribbleDoc.cpp, add the following **#include** statement:

```
#include "ScribbleItem.h"
```

Note The short filename for this header file is ScribItm.h. If you're starting from the sample source files from Scribble Step 6 to complete this tutorial step, be sure to use this short filename. Otherwise, you will not be able to compile your project.

Now you'll add the `OnGetEmbeddedItem` function override. (AppWizard provided this code in the Scratch version of Scribble you generated with Full-server support.)

3 In ClassView, point your mouse at the `CScribbleDoc` class icon and click the right mouse button.

4 From the pop-up menu, choose Add Function.

5 In the Function Type box, specify: COleServerItem*

6 In the Function Declaration box, type: `OnGetEmbeddedItem()`

7 In the Access group box, select the Protected radio button.

8 Select the Virtual check box, (note that this adds the virtual keyword to the declaration) and click OK.

9 Implement the function as follows:

```
    // OnGetEmbeddedItem is called by the framework to get the
COleServerItem
    //  that is associated with the document.  It is only called when
necessary.

    CScribbleItem* pItem = new CScribbleItem(this);
    ASSERT_VALID(pItem);
    return pItem;
```

10 For convenience you'll also provide a type-safe function to return a pointer to the specific **COleServerItem**-derived class, `CScribbleItem`, by adding the following code to the public Attributes section of ScribbleDoc.h:

```
    CScribbleItem* GetEmbeddedItem()
    { return (CScribbleItem*)COleServerDoc::GetEmbeddedItem(); }
```

Analyze OLE Server Code in InitInstance

The code below shows Scribble.cpp with the default OLE server code provided by AppWizard. (Note that the default classnames differ from those you customized for Scribble.)

```
...
#include "IpFrame.h"
...
/////////////////////////////////////////////////////////////////////////////
// The one and only CScribbleApp object

CScribbleApp theApp;

// This identifier was generated to be statistically unique for your app.
// You may change it if you prefer to choose a specific identifier.

// {9936A203-C918-11CE-B9D1-08002B321D20}
static const CLSID clsid =
{ 0x9936a203, 0xc918, 0x11ce, { 0xb9, 0xd1, 0x8, 0x0, 0x2b, 0x32, 0x1d, 0x20 } };

/////////////////////////////////////////////////////////////////////////////
// CScribSvrApp initialization

BOOL CScribSvrApp::InitInstance()
{
    // Initialize OLE libraries
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }
```

```
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();         // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();   // Call this when linking to MFC statically
#endif

    LoadStdProfileSettings();  // Load standard INI file options (including MRU)

    // Register the application's document templates.  Document templates
    //  serve as the connection between documents, frame windows and views.

    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
        IDR_SCRIBSTYPE,
        RUNTIME_CLASS(CScribSvrDoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CScribSvrView));
    pDocTemplate->SetServerInfo(
        IDR_SCRIBSTYPE_SRVR_EMB, IDR_SCRIBSTYPE_SRVR_IP,
        RUNTIME_CLASS(CInPlaceFrame));
    AddDocTemplate(pDocTemplate);

    // Connect the COleTemplateServer to the document template.
    //  The COleTemplateServer creates new documents on behalf
    //  of requesting OLE containers by using information
    //  specified in the document template.
    m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);

    // Register all OLE server factories as running.  This enables the
    //  OLE libraries to create objects from other applications.
    COleTemplateServer::RegisterAll();
        // Note: MDI applications register all server objects without regard
        //  to the /Embedding or /Automation on the command line.

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
```

```
    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // Check to see if launched as OLE server
    if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
    {
        // Application was run with /Embedding or /Automation.  Don't show the
        //  main window in this case.
        return TRUE;
    }

    // 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_INPLACE_SERVER);

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    // The main window has been initialized, so show and update it.
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();

    return TRUE;
}
```

Here is an explanation of the preceding code that was provided by AppWizard. All MFC OLE applications—whether container, server, or automation server—must call **AfxOleInit** and the static function **COleTemplateServer::RegisterAll** from the application's `InitInstance` to initialize framework support for OLE. For an application that is not an OLE server, the **CDocTemplate** object coordinates the creation of the frame window, view, and document object for the stand-alone application. The **CDocTemplate** uses menu, accelerator, and string resources passed to the constructor in `InitInstance` code, to determine the menu, accelerators, and Windows shell registration of the stand-alone application.

In the case of an OLE server application, additional information is needed. The `InitInstance` function passes this information as parameters to the **CDocTemplate::SetServerInfo** function, before it calls **CWinApp::AddDocTemplate**. Here is a description of the parameters:

```
pDocTemplate->SetServerInfo(
    IDR_SCRIBBTYPE_SRVR_EMB, IDR_SCRIBBTYPE_SRVR_IP,
    RUNTIME_CLASS(CInPlaceFrame));
```

- `IDR_SCRIBBTYPE_SRVR_EMB` is the common ID of the menu and accelerator resources loaded when Scribble is fully opened by the container application when it edits an embedded item.

- IDR_SCRIBBTYPE_SRVR_IP is the common ID of the menu, accelerators, and toolbar bitmap resources that are loaded when Scribble is activated in place in the container application. The purpose and design of these resources specifically for the in-place activated server application, as well as the purpose and design of the above resources for the fully opened server application, are explained later in this tutorial.
- RUNTIME_CLASS(CInPlaceFrame) is the **COleIPFrameWnd**-derived class provided by AppWizard. This class defines the behavior of the window created by the framework on behalf of the server application when it is activated in place by the container. The AppWizard-provided implementation of this class adds the resize border to the in-place window so that the user can resize the object while it is activated in place.

The following code defines the OLE class ID for the Scribble application and registers the application. AppWizard provides a default ID that is randomly generated. The call to **COleTemplateServer::ConnectTemplate** registers the class ID with Windows.

Note The **clsid** below will differ from the unique one that is provided when you run AppWizard.

```
static const CLSID BASED_CODE clsid =
{ 0x0002180f, 0x0, 0x0, { 0xC0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x46 } };

...

m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);
```

If the application was spawned by OLE as an in-place server or automation server, then InitInstance returns before performing additional initialization tasks that are appropriate only for stand-alone applications:

```
    if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
    {
            return TRUE;
    }
```

Framework Support for Registering the Application with Windows

An MFC OLE server can use **COleTemplate::UpdateRegistry** to register the application as an OLE server. The following AppWizard-provided code is optional.

```
// 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_INPLACE_SERVER);
```

This code enables the user to run the application once to register it as an OLE server. The preferred method of registering your application with Windows is to use one of the methods described earlier, in "Registering an OLE Server Application With Windows."

- Manually merge the SCRIBBLE.REG registration file into the system registry by using REGEDIT

  –or–

- Programmatically merge the registry information as part of your application's setup program. There are several ways you can accomplish this, as described in the topic mentioned.

Editing OLE-Related Resources

The next overall task is to edit the OLE-related resources in your server application. In general, this consists of the following steps:

- Add OLE standard resources
- Add menus
- Add toolbar buttons
- Add accelerators for in-place active or fully-opened servers

Add OLE Standard Resources

▶ To add OLE standard resources

- From the View menu, choose the Resource Includes command.

  –or–

1. In ResourceView, point the mouse cursor at the Scribble Resources folder, and click the right mouse button.
2. From the pop-up menu, choose Resource Includes.
3. In the Compile-Time Directives list box, type the following:

```
#include "afxolesv.rc"  // OLE server resources
```

This **#include** statement takes care of including some string resources referred to by the framework OLE classes. AppWizard adds this **#include** statement to your application resource file if you choose the Mini-Server, Full-Server, or Container Server option. If you look at SCRATCH\Scribble.rc, you will see that AppWizard has added this same compile-time directive.

4 Click OK to accept the changes you made in the Resource Includes dialog box.

5 Click OK when the following message box appears:

```
Directive text will be written verbatim into your resource script and
may render it incompatible
```

Add OLE Menu Resources

A server application shows different menus, depending on whether it is running stand-alone, embedded, or in-place activated. AppWizard provides three different menus for these cases:

- `IDR_SCRIBBTYPE` is the menu for the document when it is opened in the usual way.
- `IDR_SCRIBBTYPE_SRVR_EMB` is the menu for the document when the server is opened fully from the container application.
- `IDR_SCRIBBTYPE_SRVR_IP` is the menu for the document when it is activated in place (thus, "IP") in a container. When an object is activated in place, OLE merges the menu of the container application with the menu provided by the server application. The merging of the two menus is based on separator bars. Scribble's in-place menu (`IDR_SCRIBBTYPE_SRVR_IP`) looks like this:

  Edit | Pen | Help

  An example of a container's menu is seen in Container (the OLE Container tutorial discussed in Chapters 13 through 15), as shown below:

  File | | Window

  OLE merges the two menus to create the following menu when the Scribble object is activated in place in the Container application:

  File Edit Pen Window Help

  OLE merges pop-up menus from left to right in the following order:

  1. Container's pop-up menu(s) before the first separator.
  2. Server's pop-up menu(s) before the first separator.
  3. Container's pop-up menu(s) between the first and second separators.
  4. Server's pop-up menu(s) between the first and second separators.
  5. Container's pop-up menu(s) after the second separator.
  6. Server's pop-up menu(s) after the second separator.

Scribble's two distinct OLE-related menu resources are referred to in the following code in `CScribbleApp::InitInstance`:

```
pDocTemplate->SetServerInfo(
    IDR_SCRIBBTYPE_SRVR_EMB, IDR_SCRIBBTYPE_SRVR_IP,
    RUNTIME_CLASS(CInPlaceFrame));
```

AppWizard provides all the code and resources described above, except for the Pen pop-up menu, which is application specific. If you are creating a new application from scratch, each new pop-up menu must be added to each of the three resources. This is easy to do with the drag and copy feature of the menu editor. You can copy a resource by dragging it to the desired location while pressing the CTRL key, and then releasing the mouse button.

In this part of the tutorial, you will:

- Copy the two new resources, `IDR_SCRIBBTYPE_SRVR_EMB` and `IDR_SCRIBBTYPE_SRVR_IP`, from the AppWizard-provided SCRATCH\Scribble.rc to your Scribble.rc.
- Add the Pen pop-up menu to each of the menus.

▶ To copy menu resources

1. Switch to ResourceView and expand the Scribble Resources folder, then expand the Menu folder.
2. From the File menu, choose Open and select SCRIBBLE\SCRATCH\Scribble.rc. Again, expand the Menu folder.
3. Copy the menus `IDR_SCRIBBTYPE_SRVR_EMB` and `IDR_SCRIBBTYPE_SRVR_IP` from the scratch resource file to your Scribble project .RC file.
4. Close the scratch version of Scribble.rc.

▶ To copy menu items

1. Open the menu resources `IDR_SCRIBBTYPE` and `IDR_SCRIBBTYPE_SRVR_EMB` in your Scribble project resource file.
2. Arrange the two menus so they don't overlap and yet you can see both of them in the Menu editor window.
3. Copy the Pen menu from `IDR_SCRIBBTYPE` into the `IDR_SCRIBBTYPE_SRVR_EMB` menu so that it is between the Edit and View menus, as in `IDR_SCRIBBTYPE`.
4. Copy the Clear All menu item from the Edit menu in `IDR_SCRIBBTYPE` to the Edit menu in `IDR_SCRIBBTYPE_SRVR_EMB`.
5. Close `IDR_SCRIBBTYPE_SRVR_EMB` and repeat steps 1 to 3 for `IDR_SCRIBBTYPE_SRVR_IP`, placing the Pen menu as follows:

   Edit | Pen | View | | | | Help

   (Don't copy the Clear All menu item.)
6. Save and then close the menu resources.

Add OLE Toolbar Resources

The server application also provides an in-place toolbar to the container. This toolbar typically supports a different set of commands than does the container's usual toolbar, generally a subset. For example, the in-place toolbar does not support File menu commands, because the container, not the server, must handle those commands even if the server is activated in place.

AppWizard provides a default in-place toolbar resource with five buttons. The **COleIPFrameWnd**-derived class is responsible for specifying the mapping of commands to toolbar buttons.

▶ To view the default in-place toolbar

1. Open the scratch version of Scribble.rc, expand the Toolbar folder and double-click `IDR_SCRIBBTYPE_SRVR_IP`.

   The Toolbar editor opens.

2. Choose Properties from the Edit menu to view the Toolbar Button Properties page, which displays the default resource IDs and prompt strings.

3. Close the toolbar editor for the next procedure.

The Developer Studio toolbar editor makes it easy to edit your application toolbar resources. In the following procedure, you'll copy the AppWizard-generated toolbar resources for in-place activation of Scribble into your Scribble project. Then you'll edit the toolbars to match your other Scribble project toolbar resources.

▶ To copy the toolbar buttons

1. With the scratch Scribble.rc file open and the Toolbar folder expanded, switch to ResourceView in your Scribble project, and expand the Toolbar folder.

2. Hold down the CTRL key while you drag the toolbar resource, `IDR_SCRIBBTYPE_SRVR_IP`, from the AppWizard-provided scratch resource file into your Scribble project Toolbar folder.

   You may want to close the scratch Scribble.rc file at this point, to eliminate clutter in the editor window.

3. In ResourceView, double-click both Scribble toolbar resources — `IDR_MAINFRAME` and `IDR_SCRIBBTYPE_SRVR_IP` — arranging them in the editor window so they don't overlap.

4. Copy the Pen Width button from the **IDR_MAINFRAME** toolbar to the `IDR_SCRIBBTYPE_SRVR_IP` toolbar, just after the Paste button. (Hold down CTRL while you drag the button.)

5 Delete the Cut, Copy and Paste buttons from the IDR_SCRIBBTYPE_SRVR_IP toolbar by dragging them off the toolbar.

   Your in-place toolbar should now contain just three buttons: the application Help button, the Pen Widths button, and the Help Mode button. This is the toolbar that will appear when Scribble is activated as an OLE server application.

6 Save your Scribble project .RC file, and close the Toolbar editors.

For more information about working with toolbar resources, see Chapter 11, "Using the Toolbar Editor," in the *Visual C++ User's Guide*.

Add Accelerator Resources for In-Place Active or Fully-Opened Servers

Just as a server application offers different menus when it is in-place active or fully opened versus running stand-alone, the server application offers a different set of accelerators. AppWizard provides two additional accelerator resources, IDR_SCRIBBTYPE_SRVR_EMB and IDR_SCRIBBTYPE_SRVR_IP, just as it creates two additional menu resources with these same identifications. You can copy these accelerator resources from the scratch version of Scribble.rc.

▶ To copy accelerator resources

1 Just as you have done in previous procedures to copy resources, expand the Accelerator folders from both the scratch version of Scribble.rc, and your Scribble project .RC file.

2 Copy accelerator resources IDR_SCRIBBTYPE_SRVR_EMB and IDR_SCRIBBTYPE_SRVR_IP from the scratch version of Scribble.rc to your Scribble project Accelerator folder.

   You can copy both resources simultaneously, as the Resource Editors support multiple selection.

3 Save your changes to Scribble.rc, and close the scratch .RC file.

Adding Application-Specific Server Support

To complete Scribble Step 7, and implement Scribble as a fully-functional OLE server application, these are the final steps you'll take:

- Add application-specific server support to the document class.
- Implement the server item.
- Implement OLE in-place support in the view class.

Add Application-Specific Server Support to the Document Class

To finish adding application-specific server support in the document class, you have to:

- Notify the OLE server that the embedded item has moved or changed size (see next procedure).
- Change the initial size of the document.
- Implement the document's support for putting a link format on the Clipboard.

In the following procedure, you'll use the Add Member Function command to implement the `OnSetRectItems` function for class `CScribbleDoc`. The framework calls **OnSetItemRects** when the position or size of the embedded item has changed in the container, or when the clipping of the embedded item has changed in the container. Because Scribble's view is a **CScrollView**, you need to call **CScrollView::SetScrollSizes** to reflect the change in the size of the item. Because there are multiple places where the logic associated with **SetScrollSizes** must be performed, you will later write a helper function, `CScribbleView::ResyncScrollSizes`, which you will call from the override of **OnSetItemRects**.

▶ To notify the OLE server when the embedded item moves or changes size

1. In ClassView, point the mouse cursor at `CScribbleDoc` and click the right mouse button to invoke the pop-up menu.
2. Choose the Add Function command.

   The Add Member Function dialog appears.
3. Fill in the dialog box as follows:
   - In the Function Type box, type void.
   - In the Function Declaration box, type the following:

     ```
     OnSetItemRects(LPCRECT lpPosRect, LPCRECT lpClipRect)
     ```
   - In the Access area, select Protected.
4. Click OK.
5. In ScribbleDoc.cpp, implement the skeletal definition with the following code:

   ```
   // call base class to change the size of the window
   COleServerDoc::OnSetItemRects(lpPosRect, lpClipRect);

   // notify first view that scroll info should change
   POSITION pos = GetFirstViewPosition();
   CScribbleView* pView = (CScribbleView*)GetNextView(pos);
   pView->ResyncScrollSizes();
   ```

6 Add the following **#include** statement to ScribbleDoc.cpp, because the above implementation refers to `CScribbleView`:

```
#include "ScribbleView.h"
```

The short filename is ScribVw.h.

The next step is to modify `CScribbleDoc::InitDocument` so Scribble's fixed document size changes from 8 by 9 inches, to 2 by 2 inches. The current size, 8 by 9 inches, is too large for most containers.

▶ To change the initial size of the document

1 Use ClassView to jump to `CScribbleDoc::InitDocument`, and modify the comments and the parameters that return the default document size, as follows:

```
//default document size is 200 x 200 screen pixels
...
m_sizeDoc = CSize(200, 200);
```

2 Use ClassView to jump to the `CScribbleDoc` constructor, and replace the //TODO comments with the following line:

```
m_sizeDoc = CSize(200, 200);
```

The `m_sizeDoc` is also initialized in the helper member function, `InitDocument`, when a document is newly created or reopened in a stand-alone running instance of Scribble. `InitDocument` is called by Scribble's overrides of **CDocument::OnNewDocument** and **OnOpenDocument**. When Scribble is run as a server, the `OnNewDocument` and `OnOpenDocument` functions are not called. Therefore a good place to initialize `m_sizeDoc` is in the constructor.

Finally, in the following procedure you'll implement the ability for containers to execute the Paste Link command on the server's Edit menu.

▶ To implement the document's support for putting a link format on the Clipboard

1 With ScribbleDoc.cpp displayed in the editor window, from the WizardBar Object IDs list, select **ID_EDIT_COPY**, and from the Messages list select COMMAND.

2 In the Add Member Function dialog, accept the default name OnEditCopy.

3 Replace the highlighted comment with the following code:

```
CScribbleItem* pItem = GetEmbeddedItem();
pItem->CopyToClipboard(TRUE);
```

The framework function **COleServerDoc::CopyToClipboard** does all the work.

Implement the Server Item

AppWizard has done most of the work associated with implementing the server item by providing the **COleServerItem**-derived class in ScribbleItem.cpp. All you have to do is add the application-specific implementation.

The server item's `OnDraw` is called when the server document needs to draw itself as an inactive embedded object inside the container window. In contrast, the view's `OnDraw` is called when the document is activated in place inside the container. `CScribbleItem::OnDraw` needs to do essentially the same drawing that `CScribbleView::OnDraw` does. If your `OnDraw` code in your view class is complex, you will probably want to reuse that code by having your client item's `OnDraw` call a shared draw routine. In Scribble, the `CStroke::DrawStroke` routine is reused.

Recall that you previously added the AppWizard-generated file, ScribbleItem.cpp, to the project. Now you'll modify the AppWizard-provided code to support Scribble-specific drawing functions.

▶ To implement the OLE item's OnDraw function

1 Use ClassView to jump to `CScribbleItem::OnDraw`, and modify the AppWizard-provided stubbed version of the function so that it looks like the following:

```
BOOL CScribbleItem::OnDraw(CDC* pDC, CSize& rSize)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    pDC->SetMapMode(MM_ANISOTROPIC);
    CSize sizeDoc = pDoc->GetDocSize();
    sizeDoc.cy = -sizeDoc.cy;
    pDC->SetWindowExt(sizeDoc);
    pDC->SetWindowOrg(0,0);

    CTypedPtrList<CObList, CStroke*>& strokeList
        = pDoc->m_strokeList;
    POSITION pos = strokeList.GetHeadPosition();
    while (pos != NULL)
    {
        strokeList.GetNext(pos)->DrawStroke(pDC);
    }

    return TRUE;
}
```

This code sets the window extent to the size of the document so that when the document is drawn in the in-place window, the drawing will stretch to the size of the window. It is necessary to reverse the sign of the y dimension to reflect the fact that strokes' positions are maintained in **MM_LOENGLISH** coordinates by using negative y coordinates.

2 Similarly, use ClassView to jump to the AppWizard-provided version of `CScribbleItem::OnGetExtent`, and modify it as follows:

```
BOOL CScribbleItem::OnGetExtent(DVASPECT dwDrawAspect, CSize& rSize)
{
    // This implementation of CScribbleItem::OnGetExtent only handles
```

```
    // the "content" aspect indicated by DVASPECT_CONTENT.

    if (dwDrawAspect != DVASPECT_CONTENT)
        return COleServerItem::OnGetExtent(dwDrawAspect, rSize);

    // CScribbleItem::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.
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    rSize = pDoc->GetDocSize();
    CClientDC dc(NULL);

    // use a mapping mode based on logical units
    //  (we can't use MM_LOENGLISH because MM_LOENGLISH uses
    //  physical inches)
    dc.SetMapMode(MM_ANISOTROPIC);
    dc.SetViewportExt(dc.GetDeviceCaps(LOGPIXELSX),
        dc.GetDeviceCaps(LOGPIXELSY));
    dc.SetWindowExt(100, -100);
    dc.LPtoHIMETRIC(&rSize);

    return TRUE;
}
```

3 Save ScribbleItem.cpp.

The framework calls the virtual **COleServerItem::OnGetExtent** member function when the item needs to set the viewport and window extents of the server item window when the item is in-place active. The new code modifies the original code provided by AppWizard, which sets the size of the server item to an arbitrary fixed value. The function must return the size of the server item in **HIMETRIC** units. In Scribble, the server item needs to return the size of the document. Scribble stores the drawing size in physical **MM_LOENGLISH** units. For reasons explained in the next section, Scribble needs to convert the drawing size to **HIMETRIC** units based on its logical **MM_LOENGLISH** size rather than physical **MM_LOENGLISH** size.

Implement OLE In-Place Support in the View Class

To implement OLE in-place support in the view class, you must:

- Calculate logical **MM_LOENGLISH** sizes rather than physical **MM_LOENGLISH** sizes.
- Adjust the scroll view's scroll bars to reflect the use of the logical mapping mode.
- Notify OLE when the embedded item changes.

Before Scribble was enhanced to be an OLE in-place editing server, it was lazy about its device context coordinates using **MM_LOENGLISH**: Scribble did not adjust for how many logical pixels per inch were on the screen display. Most serious Windows applications scale their screen output because small fonts are rarely readable when displayed in their true (physical) size. Applications can adjust for the number of pixels in the logical inch by applying the kind of logic illustrated by the following code sample. This logic relies primarily on values returned by **CDC::GetDeviceCaps(LOGPIXELSX)** and **GetDeviceCaps(LOGPIXELSY)**. Now that Scribble is an OLE server, it should scale according to logical pixels per inch. Otherwise, you (and your user) will notice a difference in the scaling of a Scribble drawing when the Scribble server is fully open versus its scaling when it is displayed embedded in the container.

▶ To implement logical MM_LOENGLISH rather than physical MM_LOENGLISH

1 Open ClassWizard.

2 If necessary, choose the Message Maps tab and ensure that `CScribbleView` is chosen in the Class Name and Object IDs boxes.

3 In the Messages box, choose OnPrepareDC, and click Add Function.

4 Choose Edit Code.

5 Implement the override of `OnPrepareDC` as follows:

```
void CScribbleView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    CScribbleDoc* pDoc = GetDocument();
    CScrollView::OnPrepareDC(pDC, pInfo);

    pDC->SetMapMode(MM_ANISOTROPIC);
    CSize sizeDoc = pDoc->GetDocSize();
    sizeDoc.cy = -sizeDoc.cy;
    pDC->SetWindowExt(sizeDoc);

    CSize sizeNum, sizeDenom;
    pDoc->GetZoomFactor(&sizeNum, &sizeDenom);

    int xLogPixPerInch = pDC->GetDeviceCaps(LOGPIXELSX);
    int yLogPixPerInch = pDC->GetDeviceCaps(LOGPIXELSY);

    long xExt = (long)sizeDoc.cx * xLogPixPerInch * sizeNum.cx;
    xExt /= 100 * (long)sizeDenom.cx;
    long yExt = (long)sizeDoc.cy * yLogPixPerInch * sizeNum.cy;
    yExt /= 100 * (long)sizeDenom.cy;
    pDC->SetViewportExt((int)xExt, (int)-yExt);
}
```

In the following procedure, you'll implement the `ResyncScrollSizes` helper function mentioned previously.

▶ To adjust the scroll view's scroll bars to reflect the use of the logical MM_LOENGLISH mapping mode

1 In ClassView, point the cursor at `CScribbleView` and click the right mouse button.

2 From the pop-up menu, choose Add Function.

3 In the Add Member Function dialog, specify the following:

- In the Function Type box, type `void`.
- In the Function Declaration box, type `ResyncScrollSizes()`.
- In the Access area, select Public.

Click OK.

4 Fill in the skeletal definition with the following code:

```
CClientDC dc(NULL);
OnPrepareDC(&dc);
CSize sizeDoc = GetDocument()->GetDocSize();
dc.LPtoDP(&sizeDoc);
SetScrollSizes(MM_TEXT, sizeDoc);
```

5 Use ClassView to jump to `CScribbleView::OnInitialUpdate`, and replace the call to **CScrollView::SetScrollSizes** with the call to the helper function, `ResyncScrollSizes`:

```
void CScribbleView::OnInitialUpdate()
{
    ResyncScrollSizes();
    CScrollView::OnInitialUpdate();
}
```

In the next several steps you'll use WizardBar to add a function that updates the scroll bars appropriately when the window is sized.

6 Switch to ScribbleView.cpp in the editor.

7 In the WizardBar Object IDs listbox, select `CScribbleView`, and in the Messages listbox, select **WM_SIZE.**

8 Choose Yes to accept the default function name.

9 Replace the highlighted comment with a call to `ResyncScrollSizes`:

```
ResyncScrollSizes();      //  ensure that scroll info is up-to-date
```

10 Use ClassView to jump to the `CScribbleView` constructor, and replace the comment with the following code:

```
SetScrollSizes(MM_TEXT, CSize(0,0));
```

It is necessary to initialize the scroll sizes in the `CScribbleView` constructor to default values, so that the extent is defined before the first call to `OnPrepareDC`.

When Scribble's view finishes adding a new stroke in `CScribbleView::OnLButtonDown`, it calls the document's `UpdateAllViews` to inform other views that they need to invalidate a portion of the client area occupied by the new stroke. This notification is fine for Scribble when it is running stand-alone, but it is not adequate when Scribble is fully opened and editing an embedded object. In the latter case, Scribble needs to inform the container that the object has changed. This requires an additional call to **COleServerDoc::NotifyChanged**.

▶ To notify OLE when the embedded item changes

- Use ClassView to jump to `CScribbleView::OnLButtonUp`, and add the call to **NotifyChanged** just after the **ReleaseCapture** function call, before the final **return**:

```
pDoc->NotifyChanged();
```

Testing Scribble Server Functionality Using a Container Application

You have now completed all the work needed to convert Scribble to an OLE in-place editing server application. Build the project in the usual way, and follow the steps in the preview of Scribble found at the beginning of this chapter. In particular, remember to run Scribble once stand alone.

You have now converted Scribble to an OLE server that works as in the preview demonstration.

PART 3

The OLE Tutorials

CHAPTER 13

Creating an OLE Container

A container application is an application that can incorporate embedded or linked items into its own documents. The documents managed by a container application must be able to store and display OLE items as well as data created by the application itself. A container application must also allow users to insert new OLE items or edit existing ones.

In this tutorial you will create a simple OLE container application, Container. The Container document can hold several OLE in-place items that the user can resize and move to any place in the document. However, the Container document doesn't contain any application-specific objects. For an example of a container document that has both application-specific objects (draw objects) and OLE items, see the DRAWCLI sample under MFC Samples in Books Online.

Note This tutorial assumes that you are already familiar with Visual C++ and the basics of the Microsoft Foundation Class Library (MFC). If you are not, follow the Scribble tutorial in Chapters 2 through 11 before you begin this tutorial. The Scribble tutorial introduces important class library concepts and techniques, and demonstrates how to use the wizards and the resource editors.

Preview of the Container Application

Before you work through the steps of implementing Container, you might want to try out the completed application. This will help you appreciate OLE container functionality in general, and Container's container functionality in particular, from a user's point of view.

Other example container applications are provided on the Visual C++ distribution CD-ROM. Some of these samples were written using MFC, and others were written using the OLE SDK. You can easily install the files needed to build and examine these sample applications. For more information, see "Installing the Sample Files." For the OLE SDK samples, look under Windows SDK Samples in Books Online. For the MFC samples, look under MFC Samples in Books Online.

Finally, the best examples are those world-class applications that support OLE today. These are the applications you will ultimately want to test your application against.

Registering an OLE Server Application

Before you can preview the Container application (by running either the Step 2 executable source file, Container.exe, or the version you'll develop yourself by following the tutorial steps), you need to install and register at least one OLE server application for Container to access.

The Step 7 version of Scribble is a good sample server application. Another example is the HIERSVR sample (under \ SAMPLES \ MFC Samples \ OLE Samples in Books Online).

You can run either of these applications directly from Books Online. For more information, see "Previewing the Sample Applications."

For more information on OLE containers and servers, see "OLE Overview: Containers and Servers" in *Programming with MFC.*

▶ To register the Scribble server application

- Run Scribble.exe Step 7 directly from Books Online; or build and run Scribble.exe from the sample source files for Step 7.

  Note To run an application from Books Online, see "Previewing the Sample Applications." To install the Sample files, see "Installing the Sample Files."

  Running Scribble briefly as a stand-alone application registers it in the system registry as an OLE server.

▶ To preview Container

1. Run Container.exe Step 2 directly from Books Online; or build and run Container.exe from the sample source files for Step 2.
2. From the Edit menu, choose Insert New Object.
3. In the Insert Object dialog box, select Scribb Document from the Object Type box.
4. Click OK.

   A blank Scribble document is opened inside the Container document.

   - Notice the tracker rectangle with resize handles and dashed border that appears in the upper-left corner of the Container document. Container negotiates with the server (Scribble) to determine where to place the initial rectangle and what size to make it.
   - Notice how Container's menu has been merged with Scribble's (for example, notice the Pen menu) and how Container's toolbar is replaced by one provided by Scribble (notice the Pen toolbar button). This is part of the OLE menu merging feature, which enables you to edit the document within the container application.

5 Use the mouse to draw within the rectangle provided by the server running within the Container document.

6 Try out Scribble's menu and toolbar commands in place; for example, change the pen width.

7 Click outside the Scribble object, somewhere else in the Container document.

Notice how the Scribble server is deactivated; the dashed border and resize handles are removed. The application's caption changes back to indicate that a Container document now has the focus.

8 Click the Scribble object to select it.

The selection rectangle and resize handles are drawn again to indicate that this object has been selected. Notice that the cursor changes to a four-way arrow when it is over the structure.

9 Drag the Scribble object around and resize it.

10 From the Edit menu, choose Insert New Object to add additional OLE embedded objects.

11 If you like, run HIERSVR stand-alone. Copy an object from the HIERSVR to the Windows Clipboard.

12 Paste the object from the Clipboard into the Container document.

You have now seen two ways to initiate an embedded object: You can use the Insert New Object command on the Edit menu, or you can copy an object from the server and paste it into the container, as shown in steps 10 and 11 above.

The Tutorial Example: Container

This tutorial consists of two steps. The sample source files contain a subdirectory for each step: STEP1 and STEP2. Each step's subdirectory contains complete source files, and other files needed for the step. If you do not have these files on your local drive, you can easily install them from within Books Online. For more information, see "Installing the Sample Files."

This tutorial shows you how to develop an MFC OLE container application that allows visual (in-place) editing. In Step 1 (Chapter 14), you learn how to:

- Create a skeleton OLE container application capable of visual editing.
- Interpret the OLE container code provided by AppWizard.
- Coordinate the size of an embedded object with the server.
- Add hit testing and selection to the AppWizard-created container application.
- Implement activation of an embedded object.

- Implement tracker rectangles for resizing and moving items.
- Draw embedded and linked items.
- Delete embedded items.

In Step 2 (Chapter 15), you learn how to:

- Implement the Copy and Paste commands on the Edit menu.
- Implement smart invalidation.
- Improve coordination with the server to determine the size of contained objects.

CHAPTER 14

Implementing Basic OLE Container Features

This chapter covers Step 1 of the OLE container tutorial. You will learn how to:

- Create a skeleton OLE container application capable of visual editing.
- Interpret the OLE container code provided by AppWizard.
- Coordinate the size of an embedded object with the server.
- Add hit testing and selection to the AppWizard-created container application.
- Implement activation of an embedded object.
- Implement tracker rectangles for resizing and moving items.
- Draw embedded and linked items.
- Delete embedded items.

Creating a Skeleton OLE Container

AppWizard contains options that make it easy to create a skeleton application with OLE container functionality.

Note The following procedure describes how to enter the correct values for Container in the various AppWizard dialog boxes. Many of the dialog boxes contain choices you won't use. For more information on these choices, see Chapter 1, "Creating Applications Using AppWizard," in the *Visual C++ User's Guide*.

For more information on OLE options in AppWizard, see the article "AppWizard: OLE Support" in *Programming with MFC*.

▶ To create a skeleton OLE container-enabled application

1. From the File menu, choose New.

   The New dialog box appears.

2. Select Project Workspace.

   The New Project Workspace dialog box appears.

3 In the Name box, type `Container`.

AppWizard creates a project directory with this name under the main (root) directory specified in the Location box.

The workspace configuration file and project makefile are based on this name, in this case, Container.mdp and Container.mak, respectively.

4 In the Type list box, make sure MFC AppWizard (exe) is specified.

5 If necessary, use the Location box to specify a different root directory for the Container project files that AppWizard creates under the Container project directory.

Depending on the directory you last worked in, you may want to change to where the Location box currently points. You can use the Browse button to navigate to an existing directory, or type a directory name directly into the Location box. AppWizard creates this directory if it doesn't exist.

6 If any check boxes other than Win32 appear in the Platforms box, clear them.

7 Click Create.

AppWizard creates the project directory, and the MFC AppWizard-Step 1 dialog box appears.

8 Click Next in the dialog boxes for AppWizard Steps 1 and 2 to accept the default options.

9 In the AppWizard Step 3 dialog box, select Container, and then click Next to continue.

10 In the AppWizard Step 4 dialog box, click Advanced.

The Advanced Options dialog box appears.

11 If necessary, choose the Document Template Strings tab.

- In the File Extension box, type `ctr` without a period.

  The extension is reflected in the Filter Names box.

- Change the Mainframe caption to "Container" (optional).

  This string gets displayed in the title bar of the running application.

- In the Doc Type Name box, change "Contai" to "Contr."
- In the Filter Name box, change the entry to read "Container Files."
- Click Close, and Next to continue to Step 5.

12 Click Next again to accept the default options for AppWizard Step 5.

13 In the AppWizard–Step 6 dialog box, you'll change one of the class names from the defaults that AppWizard suggests. This is basically to create a cleaner-looking name. If you don't want to perform this step, it isn't necessary.

- Select the class `CContainCntrItem`. Change its name to `CContainerItem`. Change its header file to ContainerItem.h, and its implementation file to ContainerItem.cpp.

14 Click Finish.

15 The New Project Information dialog box appears, summarizing the settings and features AppWizard will generate for you when it creates your project.

16 Click OK in the New Project Information dialog box.

AppWizard creates the starter files for Container and opens the project.

In the next section you'll build the skeleton Container application AppWizard just created, and try out some of the features it already has, without your having written any code.

Trying Out the Newly Created OLE Container Application

AppWizard provides a skeleton OLE container application that already has a lot of the underlying architecture that most OLE container applications need. Build this new AppWizard-provided application and try it out.

Note Before you can preview the Container application, you need to install and register at least one OLE server application for Container to access.

The Step 7 version of Scribble is a good sample server application. Another example is the HIERSVR sample (under MFC Samples in Books Online).

You can run either of these applications to register them as an OLE server directly from Books Online. For more information, see "Previewing the Sample Applications."

▶ To register the Scribble server application

- Run Scribble.exe directly from Books Online, or build and run Scribble.exe from the sample source files for Step 7.

  Note To run an application from Books Online, see "Previewing the Sample Applications." To install the Sample files, see "Installing the Sample Files."

  Running Scribble briefly as a stand-alone application registers it in the system registry as an OLE server.

▶ To build and execute the newly created Container application

1 From the Build menu, choose Execute Container.exe.

2 Respond Yes when prompted with the message box that asks whether you want to build the executable.

Visual C++ builds and then executes Container.exe.

At this point, Container already has many of the features you saw in the preview of the completed Container application, but it is missing several capabilities, such as:

- Inserting more than one OLE object
- Selecting outside of the OLE object (back in the Container area of the document)

You will add this and other functionality during this tutorial. First, see for yourself the built-in capabilities of the skeleton Container application.

▶ To create a new Scribble drawing within Container

1 From the Edit menu, choose Insert New Object.

2 In the Insert Object dialog box, select the "Scribb Document" from the Object Type box.

3 Click OK.

A new Scribble object is activated in place. No additional code is required to implement the Insert New Object command on the Edit menu. For more information on in-place activation, see "Activation" in *Programming with MFC*.

Notice how Container's menu is merged with Scribble's and how Container's toolbar is replaced by one provided by Scribble. Again, this is already working so no additional code is required here.

▶ To edit the in-place activated object

1 Drag the mouse to draw in the embedded Scribble object.

2 Try out Scribble's menu and toolbar commands in place, such as the Pen Width command.

3 Click outside the Scribble object.

Nothing happens because Container does not yet support hit testing and selection. (Hit testing code determines the location of the cursor in the application.)

4 Press ESC to deactivate the Scribble object.

Notice that the user interface (ESC) for deactivating the selected OLE object is already incorporated into the skeleton AppWizard-created application.

5 Click the Scribble object.

Again, nothing happens because Container does not yet support hit testing and selection. Note, however, that the AppWizard-provided application always has its sole OLE object selected. Thus you can use the OLE verb command in the Edit menu to reactivate the Scribble object.

6 From the Edit menu, choose "Scribb Object" and from the cascading menu, choose Edit.

The Edit verb activates the object in place. (The Open verb fully opens the Scribble server.) For more information on OLE verbs, see the article "Activation: Verbs" in *Programming with MFC*.

Note that the Scribble server provides the tracker rectangle; Container provides the tracker only when the object is not activated in place.

7 Exit Container.

In summary, the AppWizard-provided container application already has a lot of OLE container functionality, but it is still missing some basics that are implemented in the remainder of this tutorial. For more information on creating a new OLE application, see the article "AppWizard: Creating an OLE Visual Editing Application" in *Programming with MFC*.

Examining AppWizard-Provided Code

The following description of most of the container-specific code provided by AppWizard will help you gain a preliminary understanding of how MFC OLE container support works. In addition, if you choose to add OLE container support to an already existing MFC application, this description will help you identify the code you need to manually add to your application.

CContainerApp

Tip Use ClassView to jump to the `CContainerApp::InitInstance` code we're about to examine.

AppWizard provides the application's `InitInstance` function as follows:

- Initializes the OLE libraries by calling **AfxOleInit**.
- Calls **CDocTemplate::SetContainerInfo** to assign the menu and accelerator resources that are used when an embedded item is activated in place. AppWizard gives the menu and accelerator resources the same identification: IDR_CONTRTYPE_CNTR_IP.

  The menu looks like this:

  File | | Window

  The two separator bars in the menu tell the framework where to insert pop-up menus provided by the server when the embedded item is activated in place.

For more information on how separator bars work, see "Menus and Resources: Menu Merging" in *Programming with MFC.*

The accelerator resource reflects the fact that fewer accelerators are provided by the container application when an embedded item is activated in place. The reason for this is that the server provides accelerators specific to the activated item.

CContainerView

The member `CContainerItem* m_pSelection` points to the currently selected OLE object. If no object is selected, its value is **NULL**.

The AppWizard-provided implementation of `OnDraw` relies on the simple assumption that there is only one object to be drawn, namely the sole `m_pSelection` object. Later this implementation is replaced with code that draws the multiple OLE client items (OLE embedded objects) contained in the document.

Tip Use ClassView to jump to the `OnDraw` member function of class `CContainerView` and examine the code.

```
void CContainerView::OnDraw(CDC* pDC)
{
...

    if (m_pSelection == NULL)
    {
        POSITION pos = pDoc->GetStartPosition();
        m_pSelection = (CContainerItem*)pDoc->GetNextClientItem(pos);
    }
    if (m_pSelection != NULL)
        m_pSelection->Draw(pDC, CRect(10, 10, 210, 210));
    }
```

The AppWizard-provided implementation of `IsSelected` returns **TRUE** if the specified **CObject** is the `m_pSelection` object. This code is used without changes for the Container application, which has a simple single-selection user interface. For an example of multiple selection, see the DRAWCLI sample application.

Tip Use ClassView to jump to the `IsSelected` member function, and examine the code.

```
BOOL CContainerView::IsSelected(const CObject* pDocItem) const
{
    // The implementation below is adequate if your selection consists
    //  of only CContainerItem 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;
}
```

OnInsertObject is the command handler for the Insert New Object command on the Edit menu. The AppWizard-provided implementation creates a standard **COleInsertDialog** object and calls up the dialog box. It then creates a **COleClientItem**-derived object and calls the **CreateItem** member function of the **COleInsertDialog** object to create the embedded object using the information specified by the user. For more information, see the articles "Containers: Client Items" and "Dialog Boxes in OLE" in *Programming with MFC*.

```
void CContainerView::OnInsertObject()
{
    // Invoke the standard Insert Object dialog box to obtain
    //  information for new CContainerItem object.
    COleInsertDialog dlg;
    if (dlg.DoModal() != IDOK)
        return;

    BeginWaitCursor();

    CContainerItem* pItem = NULL;
    TRY
    {
        // Create new item connected to this document.
        CContainerDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pItem = new CContainerItem(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 AppWizard-provided implementation of `CContainerView::OnSetFocus` changes the focus from the view to an embedded OLE item if the embedded item is currently activated in place. This is exactly the implementation needed by Container and by most container applications.

```
void CContainerView::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 AppWizard-provided implementation of `CContainerView::OnSize` determines if there is an OLE item (**COleClientItem**) currently activated in place. If so, the **COleClientItem** is notified that the clipping rectangle of the item has changed. This allows the server to know how much of the object is visible. When the size of the window changes, so does the size of the clipping rectangle. For example, HIERSVR uses this mechanism to implement scrolling and keyboard movement correctly.

```
void CContainerView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    COleClientItem* pActiveItem
        = GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->SetItemRects();
}
```

CContainerItem

Class `CContainerItem` is derived from **COleClientItem**. From the container application's perspective, a **COleClientItem** object represents an OLE embedded item, something it can draw and edit. The life of this object spans the life of the container document, as long as the particular item is embedded in the document. A container application typically creates a **COleClientItem** object in its implementation of the Insert Object command. Indeed, the implementation of `CContainerView::OnInsertObject` provided by AppWizard does create the `CContainerItem` object, as explained earlier. The application explicitly deletes a **COleClientItem** object only in certain cases, such as when the user presses DEL when this item is selected or when the entire containing document is destroyed. For more information, see the article "Containers: Client Items" in *Programming with MFC*.

The AppWizard-provided implementation of `CContainerItem::OnChange` simply calls **OnChange** in the base class, **COleClientItem**, and then, just to be safe, invalidates all views of the document.

Tip Use ClassView to jump to the `OnChange` handler function of class `CContainerItem`.

```
void CContainerItem::OnChange(OLE_NOTIFICATION nCode, DWORD dwParam)
{
    ASSERT_VALID(this);

    COleClientItem::OnChange(nCode, dwParam);

    // When an item is being edited (either in-place or fully open)
    //  it sends OnChange notifications for changes in the state of the
    //  item or visual appearance of its content.

    // TODO: invalidate the item by calling UpdateAllViews
    //  (with hints appropriate to your application)

    GetDocument()->UpdateAllViews(NULL);
        // for now just update ALL views/no hints
}
```

The framework calls **COleClientItem::OnGetItemPosition** during in-place activation when OLE needs to determine the location of the item. The AppWizard-provided implementation arbitrarily sets the rectangle of the item to (10, 10, 210, 210). Later this implementation is changed to reflect the actual position and size of the embedded item.

```
void CContainerItem::OnGetItemPosition(CRect& rPosition)
{
    ASSERT_VALID(this);

    // During in-place activation, CContainerItem::OnGetItemPosition
    //  will be called to determine the location of this item. The
    //  default implementation created from AppWizard simply returns a
    //  hard-coded rectangle. Usually, this rectangle would reflect the
```

```
    //  current position of the item relative to the view used for
    //  activation. You can obtain the view by calling
    //  CContainerItem::GetActiveView.

    // TODO: return correct rectangle (in pixels) in rPosition

    rPosition.SetRect(10, 10, 210, 210);
}
```

The framework calls **COleClientItem::OnChangeItemPosition** on behalf of a server to change the position of the in-place window, usually as a result of the server window being resized or the extent of the server window being changed. The AppWizard-provided implementation of the `OnChangeItemPosition` function calls the base class **COleClientItem::OnChangeItemPosition**, which in turn calls **COleClientItem::SetItemRects** to move or resize the item to the new position or size.

```
BOOL CContainerItem::OnChangeItemPosition(const CRect& rectPos)
{
    ASSERT_VALID(this);

    // During in-place activation CContainerItem::OnChangeItemPosition
    //  is called by the server to change the position of the
    //  in-place window. Usually, this is a result of the data in the
    //  server document changing such that the extent has changed or as
    //  a result of in-place resizing.
    //
    // The default here is to call the base class, which will call
    //  COleClientItem::SetItemRects to move the item
    //  to the new position.

    if (!COleClientItem::OnChangeItemPosition(rectPos))
        return FALSE;

    // TODO: update any cache you may have of the item's
    //rectangle/extent

    return TRUE;
}
```

Implementing the OLE Client Item Rectangle

The AppWizard-provided implementation of `CContainerItem` does most of the work needed for Contain, but some of its functionality needs to be enhanced.

▶ To implement the OLE client item rectangle

1 Declare `m_rect` in the public attributes section of ContainerItem.h:

```
CRect m_rect;  // position within the document
```

Tip Use ClassView to jump to the definition of class CContainerItem.

The AppWizard-provided implementation of `CContainerItem` assumed an arbitrary rectangle that locates the object in the container document. A **CRect** is needed to store the location and size of the object.

2 Initialize `m_rect` in the `CContainerItem` constructor in ContainerItem.cpp (you can jump to the constructor from ClassView). Replace the //TODO comment with the following code:

```
m_rect.SetRect(10, 10, 50, 50);
```

3 Use ClassView to jump to `OnGetItemPosition`, and replace its default implementation with the following code (leave the `ASSERT_VALID(this)` line):

```
// return rect relative to client area of view
rPosition = m_rect;
```

The AppWizard-provided implementation arbitrarily sets the rectangle to (10, 10, 210, 210) when requested by the framework. Because the rectangle for each `CContainerItem` item is now being tracked by **CRectTracker**, the framework's request is satisfied by returning the **CRect** member, `m_rect`.

In Container Step 2, this implementation is replaced with one that allows the server to negotiate the size of the object.

Complete the implementation of `OnChangeItemPosition` by adding the following code in place of the // TODO comment:

```
GetDocument()->UpdateAllViews(NULL);
m_rect = rectPos;

// mark document as dirty
GetDocument()->SetModifiedFlag();
```

The framework calls **COleClientItem::OnChangeItemPosition** on behalf of a server to change the position of the in-place window. Replace the AppWizard stub below with the marked lines. The `CContainerItem` updates its **CRect** `m_rect` according to the value requested by the framework. This means that the container document has changed. Thus views need to be notified and the document needs to be marked as dirty according to normal framework document/view rules.

In Container Step 2, the simple **UpdateAllViews** call is replaced with smart invalidation.

4 Serialize the **CRect** `m_rect` member variable in `CContainerItem::Serialize`.

- Replace the //TODO comment for storing code with:

```
ar << m_rect;
```

- Replace the //TODO comment for loading code with:

```
ar >> m_rect;
```

Implementing Hit Testing and Selection

The AppWizard-provided skeleton application initially supports only one embedded object. This part of the tutorial adds support for multiple objects by implementing hit testing and selection. Hit testing determines which of the multiple objects lies at a given point.

In the next two procedures, you'll use ClassView to add two helper functions to class `CContainerView`: `HitTestItems`, and `SetSelection`. These functions implement hit testing and selection.

▶ To implement hit testing

1 In ClassView, point your cursor at the `CContainerView` class icon and click the right mouse button.

2 From the pop-up menu, choose Add Function.

The Add Member Function dialog box appears.

3 Fill in the dialog box as follows:

- In the Function Type box, type `CContainerItem*`.
- In the Function Declaration box, type the following:

  ```
  HitTestItems(CPoint point)
  ```

- In the Access area, select Public.
- Click OK.

Visual C++ adds the declaration to the header file and creates a skeletal definition in the implementation file.

4 In ContainerView.cpp, type the following code to fill in the function definition for `HitTestItems`:

```
    CContainerDoc* pDoc = GetDocument();
    CContainerItem* pItemHit = NULL;
    POSITION pos = pDoc->GetStartPosition();
    while (pos != NULL)
    {
        CContainerItem* pItem = (CContainerItem*)pDoc-
>GetNextItem(pos);
        if (pItem->m_rect.PtInRect(point))
            pItemHit = pItem;
    }
    return pItemHit;// return top item at point
```

▶ To implement selection

1 Repeat steps 1 and 2 from the previous procedure to invoke the Add Member Function dialog.

2 Fill in the dialog box as follows:

- In the Function Type box, type `void`.

- In the Function Declaration box, type the following:

  ```
  SetSelection(CContainerItem* pItem)
  ```

- In the Access area, select Public.
- Click OK.

3 In ContainerView.cpp, fill in the implementation code:

```
// close in-place active item
if (pItem == NULL || m_pSelection != pItem)
{
    COleClientItem* pActiveItem
        = GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL && pActiveItem != pItem)
        pActiveItem->Close();
}
Invalidate();
m_pSelection = pItem;
```

4 Save the header and implementation files.

The above implementation is "lazy" in that it invalidates the entire client area of the view whenever the selection changes. In Container Step 2, this implementation is replaced with smarter invalidation.

Implementing Activation by Using a Mouse Click

Container has a standard user interface for selecting and activating embedded objects. A single click selects an object; a double-click activates it. If the object is selected, the user can move or resize it, or in general, manipulate the object as a whole. If the object is activated in place, the user can edit it. For more information, see "Activation" in *Programming with MFC*.

Implementing the **OnLButtonDown** handler so that a single click selects the embedded object follows this scheme:

- Call `HitTestItems` to find the `CContainerItem` at the point where the mouse was clicked.
- Set the selection to be this `CContainerItem`. Note, if no `CContainerItem` is located at the point where the mouse was clicked, nothing (**NULL**) is selected.
- If something is selected, set up a tracker rectangle (**CRectTracker**) around the selected object. A **CRectTracker** object is short lived. It exists only during the time a mouse event is being handled, or as you will see later, during the time a window is being repainted. In the case of a single click, the **CRectTracker** paints a rectangle with resize handles around the object.

If the item is clicked, **CRectTracker::Track** captures the mouse, enabling the user to drag the tracker rectangle around on the screen and to:

- Resize the item if the click was on a handle.
- Drag the item if the click was inside the rectangle.

When the user releases the mouse button, **CRectTracker** updates its public member variable, **m_rect**, which represents the new size of the object. For more information, see "Trackers" in *Programming with MFC*.

- If the user has resized the object (indicated by a value of **TRUE** being returned from **CRectTracker::Track**), update the `m_rect` of the `CContainerItem` object.

▶ To implement the OnLButtonDown mouse handler

1 With ContainerView.cpp open in the editor, use WizardBar to add the **WM_LBUTTONDOWN** handler for `CContainerView`.

2 Implement `CContainerView::OnLButtonDown` in ContainerView.cpp by replacing the \\TODO comments inside the skeleton handler code that WizardBar generates with the following code.

For now, the entire client area of the view is invalidated. Smarter invalidation is implemented in Step 2.

```
CContainerItem* pItemHit = HitTestItems(point);
SetSelection(pItemHit);

if (pItemHit != NULL)
{
    CRectTracker tracker;
    SetupTracker(pItemHit, &tracker);

    UpdateWindow();
    if (tracker.Track(this, point))
    {
        Invalidate();
        pItemHit->m_rect = tracker.m_rect;
        GetDocument()->SetModifiedFlag();
    }
}
```

The helper function `SetupTracker` sets up the styles of the tracker rectangle according to the state of the `CContainerItem` object, such as whether it has been selected.

▶ To implement the helper function CContainerView::SetupTracker

1 In ClassView, select the icon for class `CContainerView`, and click the right mouse button.

2 From the pop-up menu, choose Add Member Function.

3 Fill in the Add Member Function dialog as follows:

- In the Function Type box, type `void`.
- In the Function Declaration box, type:

```
SetupTracker(CContainerItem* pItem, CRectTracker* pTracker)
```

- In the Access area, select Public, and choose OK.

4 Implement the helper function with the following code:

```
pTracker->m_rect = pItem->m_rect;

if (pItem == m_pSelection)
    pTracker->m_nStyle |= CRectTracker::resizeInside;

if (pItem->GetType() == OT_LINK)
    pTracker->m_nStyle |= CRectTracker::dottedLine;
else
    pTracker->m_nStyle |= CRectTracker::solidLine;

if (pItem->GetItemState() == COleClientItem::openState ||
    pItem->GetItemState() == COleClientItem::activeUIState)
{
    pTracker->m_nStyle |= CRectTracker::hatchInside;
}
```

The **OnLButtonDblClick** handler needs to be implemented so that if the user double-clicks, the object is opened (**OLEIVERB_OPEN**). How the object is opened depends on whether the server supports in-place editing. If the user presses CTRL while double-clicking, the Open verb of the object should be called. Otherwise, call the primary verb, the meaning of which is determined by the server.

▶ To implement the OnLButtonDblClick mouse handler

1 Use WizardBar to add the **WM_LBUTTONDBLCLK** handler to `CContainerView`.

2 Implement `CContainerView::OnLButtonDblClk` in ContainerView.cpp by replacing the \\TODO comment inside the skeleton handler code that WizardBar generates with the following code:

```
OnLButtonDown(nFlags, point);

if (m_pSelection != NULL)
{
    m_pSelection->DoVerb(GetKeyState(VK_CONTROL) < 0 ?
        OLEIVERB_OPEN : OLEIVERB_PRIMARY, this);
}
```

Implementing Tracker Rectangles for Resizing and Moving Objects

When the user moves the cursor over a selected object, the cursor changes its shape to indicate the kind of manipulation available to the user. For example, if the cursor is over the resize handle at the upper-middle or lower-middle side of the tracker rectangle, the cursor changes to a two-way vertical arrow to indicate that the user can drag the upper or lower edge of the object. The framework's **CRectTracker** class implements this. All you need to do is call **CRectTracker::SetCursor**.

▶ To implement special cursors for the tracker

1 Use WizardBar to add a **WM_SETCURSOR** handler for `CContainerView`.

2 Replace the highlighted \\TODO comment with the following code to implement the `CContainerView::OnSetCursor` handler:

```
if (pWnd == this && m_pSelection != NULL)
{
    // give the tracker for the selection a chance
    CRectTracker tracker;
    SetupTracker(m_pSelection, &tracker);
    if (tracker.SetCursor(this, nHitTest))
        return TRUE;
}
```

Note Your code should come before, but not replace, the call to the base class (**CView**) that WizardBar adds to the skeleton handler:

```
return CView::OnSetCursor(pWnd, nHitTest, message);
```

Drawing the Embedded Objects

The AppWizard-provided implementation of `CContainerView::OnDraw` simply draws the one embedded object pointed to by `m_pSelection`. Now that Container supports multiple embedded objects, `OnDraw` must be reimplemented accordingly.

▶ To support drawing of multiple embedded objects

- Use ClassView to jump to the `OnDraw` function in class `CContainerView`. Replace the implementation provided by AppWizard with the following code (start after the `ASSERT` statement):

```
// draw the OLE items from the list
POSITION pos = pDoc->GetStartPosition();
while (pos != NULL)
{
    // draw the item
    CContainerItem* pItem = (CContainerItem*)pDoc-
>GetNextItem(pos);
    pItem->Draw(pDC, pItem->m_rect);
```

```
        // draw the tracker over the item
        CRectTracker tracker;
        SetupTracker(pItem, &tracker);
        tracker.Draw(pDC);
    }
```

Again, a **CRectTracker** object is used to draw the rectangle and possibly resize handles around the embedded object. The **CRectTracker** object lives only long enough to draw during this particular repaint. Another **CRectTracker** object was used, as you saw above, to handle a click on one of the resize handles. Yet another **CRectTracker** object was used to change the shape of the cursor when it was over one of the resize handles. Each of these **CRectTracker** objects is short lived; that is, they are automatic (local) variables of the respective Windows event handler. They were all initialized with the common code in `SetUpTracker`.

Deleting Embedded Objects

Deleting an embedded object is as simple as calling **COleClientItem::Delete** from a handler for the Clear command on the Edit menu pop up in the `IDR_CONTRTYPE` menu.

▶ To delete an embedded object

1 Use the menu editor to add a Delete command and separator to the Edit menu in the `IDR_CONTRTYPE` menu resource:

Edit

...

Paste &Special

&Delete

Assign the standard framework command ID, that is **ID_EDIT_CLEAR**, to the Delete command. Note that the command prompt is already defined for you by the framework:

```
Erase the selection\nErase
```

(The prompt doesn't show until you're out of edit mode for this menu command.)

2 Save the resource file, and if you like, close it.

3 Using the WizardBar, create a pair of **ON_COMMAND** and **ON_UPDATE_COMMAND_UI** handlers for **ID_EDIT_CLEAR** in ContainerView.cpp.

4 Implement `OnEditClear` as follows:

```
if (m_pSelection != NULL)
{
    m_pSelection->Delete();
    m_pSelection = NULL;
    GetDocument()->UpdateAllViews(NULL);
}
```

5 Implement `OnUpdateEditClear` as follows:

```
pCmdUI->Enable(m_pSelection != NULL);
```

To delete an embedded object in a container, simply call the object's **COleClientItem::Delete** function.

Building and Running Container Step 1

Build Container Step 1. When it compiles and links successfully, run the program. Here are some things to try:

- From the Edit menu, choose Insert New Object to create a new Scribble drawing within the Container document. When the Insert Object dialog box appears, select Scribb Document in the Object Type box.

  Notice that the Scribble object initially has a size of (10, 10, 50, 50), as determined by the `CContainerItem` constructor. Container does not consult the server about the initial size of the object.
- Resize the object to make it bigger and draw in the new object.
- Click outside the Scribble object. It is now properly deselected.
- Click the Scribble object.

  Selection now works, and the cursor changes to a four-way arrow over the object.
- Drag the object around, and resize it.

  Rectangle tracking is now working.

At this point you can try to insert linked items:

- Start HIERSVR stand-alone, create a file, and save it to disk. (For information on how to run a sample program from Books Online, see "Previewing the Sample Application.")
- Close HIERSVR.
- From the Container Edit menu, choose Insert New Object.

- Choose the Create From File option.
- Type the name of the HIERSVR file just created, or use the Browse button to select it.
- Choose the Link check box.
- Click OK.

You will see that the item now has a dashed border, and that double-clicking the item opens it instead of activating it, as would be expected from linked items.

This completes Step 1 of Contain. In Chapter 15, "Refining OLE Container Functionality," you will add the Copy and Paste commands to the Edit menu, implement smart invalidation, and implement better coordination with the server to determine the size of contained objects.

CHAPTER 15

Refining OLE Container Functionality

As implemented in Step 1, Container is almost fully functional as a general-purpose OLE container application, but it needs some refinement. To accomplish this, Step 2 adds the following:

- Implementation of the Copy and Paste commands on the Edit menu.
- Implementation of smart invalidation that optimizes Container to redraw only those objects that need to be redrawn, rather than redrawing all objects whenever one is changed.
- Better coordination with the server to determine the size of contained objects.

  To demonstrate why this is necessary, you will be asked to run Step 1 of Container. For that reason, you should save the Step 1 version of CONTAINER.EXE before you start working on Step 2.

You will probably need to make similar refinements in your container applications, although the details may vary.

Adding Command Handlers for Copy and Paste

AppWizard has already added the Copy and Paste menu items to Container's Edit menu, but these commands still need to be implemented. The **COleDocument** implementation already provides an **UPDATE_COMMAND_UI** handler for the Paste command. This handler enables the Paste command if there is anything on the Clipboard.

▶ To implement the Copy command

1 Open ContainerView.cpp in a text editor window.

2 From the WizardBar IDs drop-box, choose ID_EDIT_COPY.

3 Add both the **COMMAND** and **UPDATE_COMMAND_UI** handlers for **ID_EDIT_COPY**, and accept the default function names, `OnEditCopy` and `OnUpdateEditCopy`, respectively.

4 To implement the Copy command on the Edit menu, fill in the skeleton handler that WizardBar creates with the code below :

```
if (m_pSelection != NULL)
    m_pSelection->CopyToClipboard();
```

The Copy command on the Edit menu copies the contents of the current selection to the Clipboard. Implementing the Copy command is easy because the framework function **COleClientItem::CopyToClipboard** does all the work.

5 Fill in the skeleton handler for `OnUpdateEditCopy` with the following code:

```
pCmdUI->Enable(m_pSelection != NULL);
```

The **UPDATE_COMMAND_UI** handler for the Copy command enables the command if there is an active selection; otherwise, the command is disabled.

▶ To implement the Paste command on the Edit menu

1 With ContainerView.cpp still open in the editor, from the WizardBar IDs drop-down, choose **ID_EDIT_PASTE**.

2 From the Messages drop-list, choose **COMMAND** and accept the default name of `OnEditPaste` to create the skeleton handler.

3 Implement the Paste command with the following code:

```
CContainerItem* pItem = NULL;

TRY
{
    // Create new item connected to this document.
    CContainerDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    pItem = new CContainerItem(pDoc);
    ASSERT_VALID(pItem);

    // Initialize the item from clipboard data
    if (!pItem->CreateFromClipboard())
        AfxThrowMemoryException();  // any exception will do
    ASSERT_VALID(pItem);

    // update the size before displaying
    pItem->UpdateFromServerExtent();

    // set selection to newly pasted item
    SetSelection(pItem);
    pItem->InvalidateItem();
}
CATCH(CException, e)
{
    if (pItem != NULL)
```

```
        {
            ASSERT_VALID(pItem);
            pItem->Delete();
        }
        AfxMessageBox(IDP_FAILED_TO_CREATE);
    }
    END_CATCH
```

The Paste command on the Edit menu is somewhat like the Insert New Object command on the Edit menu in that it creates a new **COleClientItem** object. Compare the above implementation of `OnEditPaste` with the one that AppWizard provided for `OnInsertObject`. Both share some code for constructing a new `CContainerItem`.

The difference is that `OnInsertObject` initializes the item based on information requested from the user by means of a **COleInsertDialog** object as shown here:

```
// 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);
```

`OnEditPaste` initializes the item from the Clipboard, using **COleClientItem::CreateFromClipboard** as shown below:

```
if (!pItem->CreateFromClipboard())
    AfxThrowMemoryException();
```

Using Smart Invalidation

The next task in Step 2 is to implement smart invalidation. Smart invalidation involves several tasks:

- Defining the update hint
- Receiving the hint and invalidating the view
- Centralizing the sending of the update hint
- Invalidating selected and deselected objects
- Invalidating an object moved by the server
- Invalidating the tracked object

Define the Update Hint

The first task is to define the update hint.

▶ To define the update hint

- Add the following two **#define** values to ContainerDoc.h, prior to the class declaration:

```
#define HINT_UPDATE_WINDOW          0
#define HINT_UPDATE_ITEM            1
```

The two **#define HINT_** values are used for the **LPARAM** *lHint* value passed to `CContainerView::OnUpdate`, which you'll create in the next step. The first hint value, **HINT_UPDATE_WINDOW**, has the framework's default *lHint* value of 0, which means "no hint": in other words, it is an instruction to invalidate the entire client area of the view. The second, **HINT_UPDATE_ITEM,** is used to invalidate the rectangle of the view's client area occupied by the **COleClientItem** object. That rectangle is passed to `OnUpdate` using the *pHint* parameter.

Receive the Hint and Invalidate the View

The framework provides a mechanism for invalidating portions of a view by using the *lHint* and *pHint* parameters of **CView::OnUpdate**. This "update hint" mechanism is described in the Scribble tutorial and is used in Container.

▶ To receive the hint and invalidate the view

1. With ContainerView.cpp open in the editor, and CContainerView selected in the WizardBar Object IDs drop-list, select OnUpdate in the Messages drop-list to create the skeleton handler.
2. Accept the default handler name.
3. Implement **OnUpdate** with the following code:

```
switch (lHint)
{
case HINT_UPDATE_WINDOW: // invalidate entire window
    Invalidate();
    break;
case HINT_UPDATE_ITEM:      // invalidate single item
    {
        CRectTracker tracker;
        SetupTracker((CContainerItem*)pHint, &tracker);
        CRect rect;
        tracker.GetTrueRect(rect);
        InvalidateRect(rect);
    }
    break;
}
```

The rectangle to be invalidated for **HINT_UPDATE_ITEM** should include the area that might be occupied by a tracker around the object. The implementation of `CContainerView::OnUpdate` takes this into account.

Centralize the Sending of Update Hints

There are several occasions when Container needs to send the **HINT_UPDATE_ITEM** hint for a `CContainerItem` object, for example, when the object is selected, deselected, resized, or otherwise changed. The **HINT_UPDATE_ITEM** hint must be passed to **OnUpdate** in conjunction with the **CObject*** *pHint* parameter, which is a pointer to `CContainerItem`. Thus, it makes sense to have the `CContainerItem` object itself send the update hint by using the document's `UpdateAllViews` function. `CContainerItem::InvalidateItem` is a helper function that you can call whenever you want to send the hint.

▶ To centralize the sending of update hints in the CContainerItem object

1. In ClassView, point the cursor at the icon for class `CContainerItem`, and press the right mouse button to invoke the pop-up menu for this class.
2. Choose Add Function.
3. In the Add Member Function dialog:
   - In the Function Type edit box, type `void`.
   - In the Function Declaration edit box, type:
     `InvalidateItem()`
   - In the Access area, select Public, and click OK.
4. To implement `CContainerItem::InvalidateItem`, type the following code inside the skeleton handler:

```
GetDocument()->UpdateAllViews(NULL, HINT_UPDATE_ITEM, this);
```

Note that the framework keeps track of which document object owns the `CContainerItem` object and therefore implements `CContainerItem::GetDocument`.

Invalidate Selected and Deselected Objects

To eliminate unnecessary repainting whenever the user changes the selected object, only the old and the new selected objects need to be invalidated. This results in smarter repainting than simply invalidating the entire client area of the view.

▶ To add update hints to selection code

- Use ClassView to jump to the `SetSelection` member function in `CContainerView`, and replace the following code:

```
Invalidate();
m_pSelection = pItem;
```

with this code:

```
// update view to new selection
if (m_pSelection != pItem)
{
    if (m_pSelection != NULL)
        OnUpdate(NULL, HINT_UPDATE_ITEM, m_pSelection);

    m_pSelection = pItem;
    if (m_pSelection != NULL)
        OnUpdate(NULL, HINT_UPDATE_ITEM, m_pSelection);
}
```

Invalidate Tracked Object

When the user clicks an object, the tracker needs to be drawn around the newly selected object. This code invalidates the extra area occupied by the tracker.

▶ To add update hints to OnLButtonDown

- Use ClassView to jump to the implementation of `CContainerView::OnLButtonDown`, and replace:

```
Invalidate();
pItemHit->m_rect = tracker.m_rect;
```

with the following code:

```
pItemHit->InvalidateItem();
pItemHit->m_rect = tracker.m_rect;
pItemHit->InvalidateItem();
```

Invalidate Object Moved by the Server

The framework calls **COleClientItem::OnChangeItemPosition** whenever the server requests a change in the position of the in-place activated object. This is one of several occasions for which you can implement smart repainting.

▶ To send an update hint when the position of the CContainerItem object changes

- Use ClassView to jump to the implementation of `CContainerItem::OnChangeItemPosition`, and replace:

```
GetDocument()->UpdateAllViews(NULL);
m_rect = rectPos;
```

with the following code:

```
InvalidateItem();
m_rect = rectPos;
InvalidateItem();
```

Coordinating with the Server to Determine Size of Object

The following exercise demonstrates why Container needs to coordinate better with servers to determine the size of embedded objects. For more information on OLE containers and servers, see the article "OLE Overview: Containers and Servers" in *Programming with MFC*.

▶ To demonstrate why coordinating with the server is necessary

1. If you have not already run the HIERSVR sample, do so now. Run HIERSVR once to register this OLE server application with OLE and then close it. (For information on running sample applications directly from Books Online, see "Previewing the Sample Applications.")
2. Similarly, run Step 1 of Container, but leave it running. (You can either use your version, or the sample source version.)
3. From the Container Edit menu, choose Insert New Object.
4. Choose MFC Hierarchy List as the object type.

   Notice that the initial size of the HIERSVR object is (10, 10, 50, 50), as determined by the `CContainerItem` constructor. Container does not give HIERSVR the opportunity to set the initial size of the object.
5. From HIERSVR's Edit menu, choose Add Node to add a second node.

   Fill in a name for the node; the other options in the Add Node dialog box don't matter for the purposes of this demonstration.

   Notice that Container correctly increases the height of the object to accommodate the new node. You can add more nodes, and Container continues to increase the height of the object. In Step 1, the implementation of `CContainerItem::OnChangeItemPosition` changes the height of the in-place activated object at the request of HIERSVR.
6. Deactivate the HIERSVR object, then click once to select it.
7. From the Edit menu, choose Hierarchy List Object, then choose Open from the submenu.

   This fully opens the HIERSVR server application. Arrange HIERSVR and Container on the screen so you can see both applications at the same time.
8. In HIERSVR, from the Edit menu choose Add Node to add another node.

   Notice that the size of the object in Container does not change to accommodate the new node. Rather, it stays the same size and compresses the nodes using a smaller font, so that the *N*+1 nodes now occupy the same space as the original *N* nodes. Add more nodes, and they become more and more compressed in the same space in the container.

9 Close both applications.

What is happening here? Why does `OnChangeItemPosition` change the size of the in-place window when a new HIERSVR node is added, but not if it is being updated when HIERSVR is running fully opened?

The reason is that `OnChangeItemPosition` is called by the framework only when the object is in-place activated. The server temporarily provides the object with its own in-place window and calls to give the container a chance to customize the size of the in-place window.

When the server is fully opened, the situation is much different (although it appears to be the same): When the server is fully opened, the object in the container is selected but not activated in place. When the user edits the fully opened object so that its natural size changes, as in the case of adding a node in HIERSVR, the server indirectly (through the framework) calls `CContainerItem::OnChange` instead of `OnChangeItemPosition`. At this time, Container needs to find out the new natural size of the object from HIERSVR. It does this by calling **COleClientItem::GetCachedExtent**.

COleClientItem::GetCachedExtent asks the server for the natural extent of the object. The natural extent is the size of the object as it would appear on the printed page (in **MM_HIMETRIC** units). In HIERSVR's case, the natural extent reflects (1) the font size that the user can specify with the Change Font command on the Tree menu, and (2) the number of nodes in the HIERSVR object.

The `CContainerItem::OnChange` function is not the only place where Container needs to call **COleClientItem::GetCachedExtent** to get the natural extent of the object and then set the `m_rect` of the `CContainerItem`. Therefore, you will implement the helper function `UpdateFromServerExtent` as described in "Get the Extent of the CContainerItem Object from the Server," following.

Get the Extent of the CContainerItem Object from the Server

To get the extent of the `CContainerItem` object from the server, and update the `m_rect` of the container item, implement the helper function `CContainerItem::UpdateFromServerExtent`.

▶ To get the extent of a CContainerItem object

1 In ClassView, point to the icon for class `CContainerItem`, and click the right mouse button.

2 From the pop-up menu, choose Add Function.

3 Fill in the Add Member Function dialog as follows:

- In the Function Type box, type `void`.
- In the Function Declaration box, type the following:

  ```
  UpdateFromServerExtent()
  ```

- Under Access, choose Public.
- Click OK.

4 In ContainerItem.cpp, implement the helper function by entering the following code:

```
CSize size;
if (GetCachedExtent(&size))
{
    // OLE returns the extent in HIMETRIC units -- we need pixels
    CClientDC dc(NULL);
    dc.HIMETRICtoDP(&size);

    // only invalidate if it has actually changed and also only
    // if it is not in-place active.
    if (size != m_rect.Size() && !IsInPlaceActive())
    {
        // invalidate old, update, invalidate new
        InvalidateItem();
        m_rect.bottom = m_rect.top + size.cy;
        m_rect.right = m_rect.left + size.cx;
        InvalidateItem();

        // mark document as modified
        GetDocument()->SetModifiedFlag();
    }
}
```

Update the CContainerItem Rectangle When the Item's Natural Extent Changes

When the fully opened server (for example, HIERSVR) notifies the container about a change (a new node) that affects the natural extent of the object, the `CContainerItem` rectangle needs to be updated.

▶ To update the CContainerItem rectangle when the item's natural extent changes

- Use ClassView to jump to the implementation of `OnChange` in `CContainerItem`, and replace the lines starting with the //TODO code, with the following code:

```
switch (nCode)
{
case OLE_CHANGED:
    InvalidateItem();
    UpdateFromServerExtent();
```

```
        break;
    case OLE_CHANGED_STATE:
    case OLE_CHANGED_ASPECT:
        InvalidateItem();
        break;
    }
```

Notice that the `CContainerItem` object has to be invalidated whenever the server sends a notification that the object has changed. The constant values that `nCode` can assume are defined by the framework.

Update the Rectangle of a Newly Inserted Object

As a user-interface design decision in Container, the rectangle of a newly inserted object is updated to reflect its natural extent, as determined by the server. A container application can ignore the natural extent if, for example, you prefer to clip the object in the rectangle.

▶ To update the rectangle of a newly inserted object to its natural extent

- Insert the following code into `CContainerView::OnInsertObject`, just before the comment `// If item created from class list ...`:

```
pItem->UpdateLink();
pItem->UpdateFromServerExtent();
```

COleClientItem::UpdateLink is called so that if the server is fully open, `Container` has a visual representation of the newly created item, even though the item hasn't been changed by the user yet.

▶ To implement smart invalidation of the newly inserted item

- In the same `OnInsertObject` function, replace the code:

```
...
    // TODO: reimplement selection as appropriate for your
    // application
    m_pSelection = pItem;
    pDoc->UpdateAllViews(NULL);
...
```

with

```
    ...
    SetSelection(pItem);
    pItem->InvalidateItem;
    ...
```

Building and Running

Build the Step 2 version of Container. It now performs exactly as you previewed it in Chapter 13.

CHAPTER 16

Creating an OLE Automation Server

An "automation server" is an application that exposes programmable objects to other applications, which are called "automation clients." Exposing programmable objects enables clients to "automate" certain functions by directly accessing those objects and using the services they make available.

Exposing objects is beneficial when applications provide functionality that is useful for other applications. For example, a word processor might expose its spell-checking functionality so that other programs can use it. Exposure of objects thus enables vendors to improve their applications by using the ready-made functionality of other applications. In this way, OLE automation applies some of the principles of object-oriented programming, such as reusability and encapsulation, at the level of applications themselves.

More important is the support OLE automation provides to users and solution providers. By exposing application functionality through a common, well-defined interface, OLE automation makes it possible to build comprehensive solutions in a single general programming language, such as Microsoft Visual Basic, instead of in diverse application-specific macro languages.

This tutorial leads you through the basic steps of implementing an OLE automation server. You will create a simple automation server application and test it using the Autodriv sample application included in Visual C++.

If you want to learn about implementing an OLE automation client application, look at the CALCDRIV sample application and its description in Samples \ MFC Samples \ OLE Samples in Books Online.

Note This tutorial assumes that you are already familiar with Visual C++ and the basics of the Microsoft Foundation Class Library (MFC). If you are not, follow the Scribble tutorial in Chapters 2 through 11 before you begin this tutorial. The Scribble tutorial introduces important class library concepts and techniques, and it demonstrates many aspects of Microsoft Developer Studio—such as ClassWizard, AppWizard, WizardBar, ClassView, and many of the built-in resource editors—that make developing your applications more streamlined and intuitive.

The Tutorial Example: AutoClik

In this tutorial you will create AutoClik, a simple OLE automation server application. When running as a stand-alone application, AutoClik does nothing but display some text at the last point at which the user clicked the mouse. The user can change the text displayed by specifying it in a dialog box. When running as an automation server, AutoClik allows automation clients to simulate both the mouse clicking and the changing of text (without bringing up the dialog box).

An automation server is not necessarily an OLE object server; AutoClik isn't. AutoClik could have been implemented as both an automation server and an OLE object server, but this tutorial focuses entirely on adding automation server functionality to an application.

How Automation Clients Access Automation Server Objects

For an automation client to drive an automation server, the client must gain knowledge of one or more "dispatch interfaces" of the server. A dispatch interface is the external programming interface of some grouping of functionality exposed by the automation server. AutoClik provides two dispatch interfaces. The first exposes AutoClik's mouse clicking and text data entry functions. The second, introduced for tutorial rather than practical reasons, represents a simple structure: a point given by x and y coordinates.

A dispatch interface consists of two types of programming interfaces: *properties* and *methods*. AutoClik exposes both. An automation client can ***get*** or ***set*** the x and y properties representing the location of the text in AutoClik's window. Or an automation client can set the x and y coordinates and the text all at once by using a method with three parameters—`x`, `y`, and `text`.

To exercise AutoClik's automation functionality, you will use the Autodriv sample application provided with Visual C++. Autodriv is a simple OLE Automation client application. The following preview of AutoClik illustrates how you can drive an automation server using Autodriv.

Preview of the AutoClik Application

Before you work through the steps of implementing AutoClik, try out the completed application. This will help you appreciate OLE automation functionality in general, and AutoClik's automation server functionality in particular, from a user's point of view.

The first step is to register AutoClik with Windows. Just as with OLE object servers, an OLE automation server must be registered before it can be driven by any automation client.

▶ To install and register the AutoClik automation server

1 Run Autoclik.exe from the sample source files for Step 3 (you can do this directly from Books Online).

Note To install the Sample files, see "Installing the Sample Files." To run an application from Books Online, see "Previewing the Sample Applications."

Running AutoClik briefly as a stand-alone application registers it in the system registry as an OLE server.

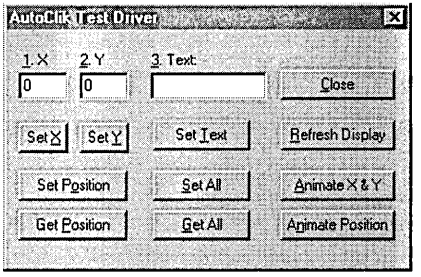
2 Close AutoClik.

▶ To preview AutoClik

1 Run Autodriv.exe from the sample source files.

This displays the AutoClik Test Driver dialog box shown in Figure 16.1.

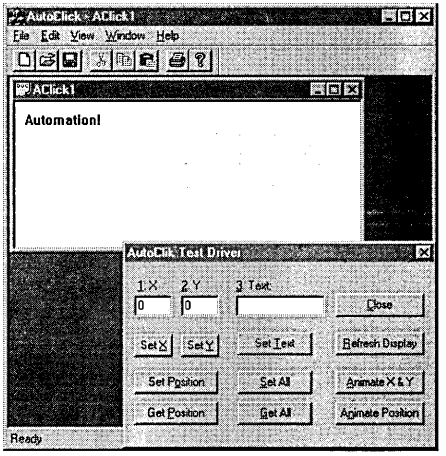
Figure 16.1 AutoClik Test Driver Dialog Box



2 Arrange the open windows on your desktop, minimizing any as necessary, so that AutoClik is visible next to Autodriv's window, without obscuring it.

Autodriv launches AutoClik on startup (Figure 16.2).

Figure 16.2 AutoClik Window Next to Autodriv Window



▶ To explore the automation server features of AutoClik

1 Note that the X, Y, and Text fields in the AutoClik Test Driver window are initially blank. To change this, click Get All.

The current AutoClik coordinates and text are now displayed in Autodriv's window.

2 Click around in the AutoClik window.

Notice that the X, Y, and Text fields in Autodriv do not change. That is because the automation is one direction between Autodriv and AutoClik. Although you can implement an automation server to notify the automation client about changes, such as the new coordinates in AutoClik, this feature requires that additional callback/notification interfaces be established so that the automation client can implement them.

3 From AutoClik's Edit menu, choose Change Text.

4 In the simple dialog box that appears, change the text to "hello", and click OK.

5 In the Autodriv window, click Get All.

It now shows new X, Y, and Text values.

6 Change the X, Y, and Text fields in Autodriv, and then click Set All.

AutoClik accepts the changes.

7 Explore other Autodriv commands:

Command	Description
Set X, Set Y	Accesses just the x or y coordinate of the text. The X and Y properties of AutoClik's document dispatch interface are exposed by using Get and Set methods. AutoClik's implementation of SetX and SetY includes updating the window to reflect the change.
Get Position, Set Position	Changes the x and y coordinates of the text by using a pointer to AutoClik's second dispatch interface, which represents a Point.
Set Text	Changes the Text property of AutoClik, which is directly exposed as a string rather than by using a pair of Get and Set functions. This means that when you choose the Set Text command, AutoClik has no opportunity to detect the change as it did when you chose the Set X, Set Y, or Set Position commands. Therefore, AutoClik does not immediately update its window. To do so, you must then choose the Refresh Display button.
Set All	Simultaneously changes AutoClik's X, Y, and Text properties through its SetAllProps method, which accepts these as three parameters. AutoClik's implementation of SetAllProps includes the immediate updating of the window.

Command	Description
Get All	Queries the `X`, `Y`, and `Text` properties of AutoClik, perhaps after you have clicked around in the AutoClik window without Autodriv's knowledge.
Refresh Display	Updates AutoClik's window based on the most recent values of `X`, `Y`, and `Text`, which might have been previously set using automation.
Animate X & Y	Updates AutoClik's x and y coordinates in 20 steps, by individually updating the `X` property and then the `Y` property. Notice that the text moves horizontally, then vertically, then horizontally, then vertically, and so on.
Animate Position	Updates AutoClik's x and y coordinates through its Point interface. Because the Point interface updates both the `X` and `Y` values at the same time, the animation results in a smooth diagonal movement of the text across AutoClik's window.

Overview of AutoClik Steps 1, 2, and 3

The AutoClik tutorial consists of three steps: STEP1, STEP2, and STEP3. The sample source files contain a subdirectory for each step and each step's subdirectory contains complete source files, and other files needed for the step. If you do not have these files on your local drive, you can easily install them from within Books Online. For more information, see "Installing the Sample Files."

In Step 1 (Chapter 17), you will learn how to:

- Create a skeleton OLE automation server using the OLE Automation option in AppWizard.
- Change the external name of the dispatch interface created by AppWizard.
- Analyze the code created by AppWizard.
- Implement AutoClik so it can run as a stand-alone application.

In Step 2 (Chapter 18), you implement the document dispatch interface. You will learn how to use ClassWizard to:

- Expose the `CAutoClickDoc` member variable `m_pt` by using the Get and Set methods in AutoClik's document dispatch interface.
- Expose the `CAutoClickDoc` member variable `m_str` as a property in AutoClik's document dispatch interface.
- Add automation methods for `RefreshWindow`, `SetAllProps`, and `ShowWindow`.

In Step 3 (Chapter 19), you will implement the second Point interface and expose AutoClik's `X` and `Y` values by using this Point interface. You will learn how to:

- Use ClassWizard to create a new **CCmdTarget**-derived class with a dispatch interface.
- Implement one dispatch interface with reference to a second dispatch interface.

CHAPTER 17

Enabling OLE Automation in an Application

In Step 1 of AutoClik, you will:

- Create a skeleton OLE automation server using the OLE Automation option in AppWizard.
- Change the external name of the dispatch interface created by AppWizard.
- Analyze the code created by AppWizard.
- Implement AutoClik so it can run as a stand-alone application.
- Build and run AutoClik.

Creating a Skeleton OLE Automation Server

The following procedure describes values you will enter into the various AppWizard dialog boxes in order to create the AutoClik skeleton files. You won't be entering a value for every option that appears in these dialog boxes. For more information on the other options, see Chapter 1, "Creating Applications Using AppWizard," in the *Visual C++ User's Guide*.

▶ To create a skeleton OLE automation server application

1. From the File menu, choose New.

   The New dialog box appears.

2. Select Project Workspace and click OK.

   The New Project Workspace dialog box appears.

3. In the Name box, type `AutoClik`.

   AppWizard creates a project directory with this name under the main (root) directory specified in the Location box.

4. In the Type list box, make sure MFC AppWizard (exe) is specified.

5 If necessary, use the Location box to specify a different root directory for the AutoClik project files that AppWizard creates under the AutoClik project directory.

Depending on the directory you last worked in, you may want to change where the Location box currently points to. You can use the Browse button to navigate to an existing directory, or type a directory name directly into the Location box. AppWizard creates this directory if it doesn't exist.

6 If any check boxes other than Win32 appear in the Platforms box, clear them.

7 Click Create.

AppWizard creates the project directory, and the MFC AppWizard–Step 1 dialog box appears.

8 Click Next in the dialog boxes for AppWizard Steps 1 and 2 to accept the default options.

For more information on the various options that appear in these dialog boxes, see Chapter 1, "Creating a New Application Using AppWizard," in the *Visual C++ User's Guide*.

9 In the AppWizard–Step 3 dialog box, select the OLE Automation check box.

10 Choose Next.

11 In the Step 4 dialog, choose the Advanced button.

The Advanced Options dialog box appears, with the Document Template Strings page tab selected.

- In the File Extension box, specify ack.

  The specified file extension is reflected in the Filter Name box.

- In the File Type ID box, edit it to read AutoClick.Document.

  This identifier is initially created by AppWizard, based on the project name. The OLE client application (such as Autodriv) uses this name to access the automation object, or dispatch interface. This name corresponds to the **regFileTypeID** string in the **CDocTemplate::GetDocString** function.

- Change the Mainframe caption to AutoClick (optional).

  This string gets displayed in the title bar of the running application.

- In the Doc Type Name box, change AutoCl to AClick (optional).

  Notice that the change is reflected in the File New Name (OLE Short Name) box as well. This name corresponds to the **fileNewName** string in **CDocTemplate::GetDocString**.

- In the Filter Name box, change AutoCl files to AutoClick Files (optional).

Note The File Type ID and Doc Type Name strings are discussed further in the next section, "Analyzing the Dispatch Interface Name."

12 Click Close, and then click Next to accept the options for the Step 4 and 5 dialogs.

In the dialog box for AppWizard–Step 6, you can check and, if necessary, change your project file and classnames. For the purposes of AutoClik, we'll just expand the classnames for clarity. (This step is optional.)

13 Optionally, in the AppWizard–Step 6 dialog box, make the following changes:

- Change classname CAutoClikApp to CAutoClickApp.
- Change classname CAutoClikDoc to CAutoClickDoc. Change the header and implementation files accordingly.
- Change classname CAutoClikView to CAutoClickView. Change the header and implementation files accordingly.

14 Chose Finish.

You may be prompted with the following message:

"A unique class ID already exists in the registration database for this document type. Use existing ID?"

This message appears if you ran the Step 3 version of AutoClik. Running AutoClik adds an entry to the registry for its document type (AutoClick.Document).

Note You can respond either Yes or No to the message dialog. If you respond No, however, and then run your new version of AutoClik, you will need to re-register AutoClik before you can run the Autodriv sample, as described in "Preview of the AutoClik Application." To re-register AutoClik, run the Step 3 source again. Running a new AutoClik application overwrites the existing registry entry and therefore invalidates the class ID Autodriv uses to identify this creatable OLE object. Responding Yes tells AppWizard not to overwrite the existing registry entry.

15 Respond as you choose to the message dialog.

The New Project Information dialog box appears, summarizing the settings and features AppWizard will generate for you when it creates your project.

You might want to take a moment to examine the application type, classes, and features that AppWizard automatically provides.

16 Click OK in the New Project Information dialog box.

AppWizard creates all necessary files, and opens the AutoClik project.

Analyzing the Dispatch Interface Name

The document template string resource is where MFC expects to find a lot of information about an application or a particular document of an application, such as default filename extensions of files saved by the application. If the application is an automation server, MFC also expects to find information specific to OLE automation.

The name of the dispatch interface is a literal string that automation clients use to access the automation server application. If you open the application's string resource, you can look at, or change, this string.

▶ To examine the document template string resource

- From ResourceView, expand the String Table folder and double-click the String Table resource.

  This opens the table of string resources for your application.

The ID for the string resource that contains the name of the dispatch interface is IDR_<Doc Type Name>TYPE, created by AppWizard, which is registered in the application's `InitInstance`. The Doc Type Name appears in the AppWizard–Step 4 Advanced Options dialog box (see step 11 of the procedure, "Creating a Skeleton OLE Automation Server.") For AutoClik, the string ID is `IDR_ACLICKTYPE`.

To view the strings for a particular resource, open the String Properties page.

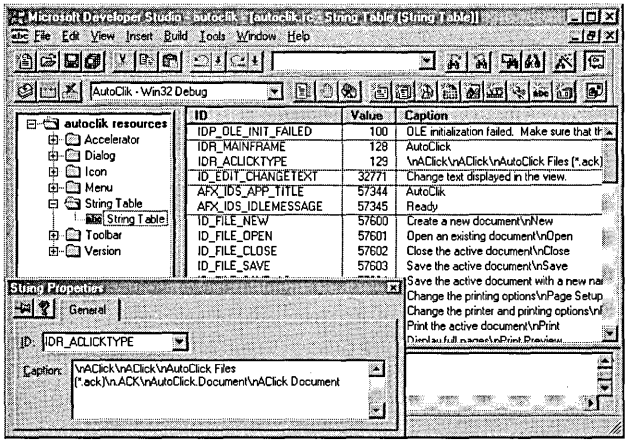
▶ To open the string properties page

- With the application string table open, choose Properties from the Edit menu.

  The String Properties window opens, with the properties for the selected string displayed.

You can see the strings for `IDR_ACLICKTYPE` in the Caption area of the String Properties dialog, as illustrated in Figure 17.1.

Figure 17.1 IDR_ACLICKTYPE in the String Editor



This string resource consists of several strings separated by newline characters (\n). It contains the following strings:

```
\nAClick\nAClick\nAutoClick Files (*.ack)\n.ACK\nAutoClick.Document\nAClick Document
```

The string "AutoClick.Document" is the name, provided by AppWizard, (and modified by you) of the automation object, or dispatch interface. You modified this value in the Step 4 Advanced Options dialog box, in the File Type ID box. Autodriv refers to this object name in the **OnCreate** function of the dialog class:

```
int CAutoDrivDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CDialog::OnCreate(lpCreateStruct) == -1)
        return -1;  // fail

    if (!m_autoClikObject.CreateDispatch(_T("AutoClick.Document")))
    {
        AfxMessageBox(IDP_CANNOT_CREATE_AUTOCLICK);
        return -1;  // fail
    }
    m_autoClikObject.ShowWindow();

    return 0;   // success
}
```

Note An automation server may have more than one automation object. AutoClik will have two automation objects. The initial AppWizard-created application has only one automation object, which is the one identified in the document template string resource described above.

Analyzing AppWizard-Provided Code

Before implementing AutoClik's basic behavior, let's look at the AppWizard-provided code that enables automation server support in AutoClik.

Application Class of an Automation Server

The work of enabling an MFC OLE Automation server application is done mostly in the `InitInstance` member function of your application's **CWinApp**-derived class. AutoClik's application class is found in AutoClik.cpp. AppWizard provides this code for you.

All MFC OLE applications require the following call to **AfxOleInit**, which initializes the OLE DLLs so they can call OLE interfaces:

```
if (!AfxOleInit())
{
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}
```

All MFC OLE automation server applications, as well as OLE object servers, require an OLE Class ID. The call to the **ConnectTemplate** member function of class **COleTemplateServer** registers the Class ID with Windows.

```
static const CLSID BASED_CODE clsid =
{ 0x2106e720, 0xaef8, 0x101a, { 0x90, 0x5, 0x0, 0xdd, 0x1, 0x8, 0xd6, 0x51 } };

...

// Connect the COleTemplateServer to the document template.
//  The COleTemplateServer creates new documents on behalf
//  of requesting OLE containers by using information
//  specified in the document template.
m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);
```

Note The numbers shown in the **clsid** line are generated at random, so the numbers in your code will most likely be different from the ones shown here.

A framework application that is an OLE automation server can use **COleTemplateServer::UpdateRegistry** to register itself as an OLE Automation server (OLE Application Type: **OAT_DISPATCH_OBJECT**). This AppWizard-provided code is optional.

```
m_server.UpdateRegistry(OAT_DISPATCH_OBJECT);
COleObjectFactory::UpdateRegistryAll();
```

Alternatively, you can register your application by using one of the two other methods described in "Creating an OLE Server":

- Manually merge the AppWizard-provided AUTOCLIK.REG registration file into the Windows registration file, using REGEDIT.
- Programmatically merge the registration as one of the tasks of your application's installation program.

Document Class of an Automation Server

When you choose the Automation Support option in AppWizard, it not only enables the application as a whole to support automation but also specifically enables the document class (in AutoClickDoc.cpp) to expose properties and methods by using automation.

The document class provided by AppWizard is derived from **CDocument**; therefore, your application's document class is derived indirectly from **CCmdTarget**. To be exposed through automation, a **CCmdTarget**-derived class must call its member function, **EnableAutomation**, from its constructor and must also include a dispatch map. Dispatch maps are like MFC message maps in that you do not edit them directly. AppWizard and ClassWizard edit them for you.

The AppWizard-provided dispatch map in the document's header file looks like this:

```
//{{AFX_DISPATCH(CAutoClickDoc)
    // NOTE - the ClassWizard will add and remove member
    //    functions here.
    //    DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_DISPATCH
DECLARE_DISPATCH_MAP()
```

and is implemented in the document's .CPP file like this:

```
BEGIN_DISPATCH_MAP(CAutoClickDoc, CDocument)
    //{{AFX_DISPATCH_MAP(CAutoClickDoc)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //     DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

As you will see in Steps 2 and 3 of AutoClik, whenever you add a new property or method, ClassWizard adds an entry to the dispatch map.

The constructor of an automated **CCmdTarget** object must call **CCmdTarget::EnableAutomation**, as implemented by AppWizard:

```
CAutoClickDoc::CAutoClickDoc()
{
    EnableAutomation();

    AfxOleLockApp();
}
```

If the automation server application supports being initially loaded by using automation, then the constructor and destructor of the document class should call **AfxOleLockApp** and **AfxOleUnlockApp**, respectively. AppWizard provides the constructor and destructor of the document class. The calls to **AfxOleLockApp** and **AfxOleUnlockApp** are required so that AutoClik gracefully terminates any interactions with automation clients before exiting.

Generally, createable objects need this. That way, if a client application creates an object of that type causing the automation server to start, the server will exit when the object goes out of scope in the client.

```
CAutoClickDoc::~CAutoClickDoc()
{
    AfxOleUnlockApp();
}
```

Creating an OLE Type Library

AppWizard adds a file named AutoClik.odl to the project. AutoClik.odl is an Object Definition Library text file. It is input to the MKTYPLIB.EXE tool which creates a type library (.TLB) file named AutoClik.tlb. The binary type library (.TLB) can be used by other applications to gain information about the automation server. This information includes a list of the automation objects provided by the automation server, and for each automation object, a list of properties and methods exposed by the automation server.

Whenever you define new automation objects, and define new methods and properties for the automation server, ClassWizard adds information to the .ODL file. When you build the application, Microsoft Developer Studio spawns MKTYPLIB.EXE to create an updated .TLB file.

The Read Type Library option of ClassWizard is a good example of how the type library file is used. ClassWizard supports not only the development of automation servers, as in this tutorial, but also the development of automation clients. An automation client accesses the properties and methods of the automation server. The Read Type Library option of ClassWizard creates a **CCmdTarget**-derived class for each automation object defined by the automation server. In the code for the automation client, you can then refer to the methods and properties of the automation server simply as C++ class member functions and member variables.

For more information on Object Definition Library and Type Library files, see the OLE documentation.

Implementing AutoClik's Basic Behavior

The rest of Step 1 implements AutoClik's basic behavior, which consists of displaying text at mouse clicks and accepting changed text by using a simple dialog box. There is nothing else relating to automation server support in this step.

▶ To add the member variables to AutoClik's document class

1 Declare the following member variables in the public Attributes section of AutoClickDoc.h:

```
CPoint m_pt;
CString m_str;
```

2 Use ClassView to jump to the `CAutoClickDoc` constructor, and add the following lines (after the **EnableAutomation** call):

```
m_pt = CPoint(10,10);
m_str = _T("Automation!");
```

3 Serialize the member variables in the document class. Use ClassView to jump to the **Serialize** member function, and implement it with the following code.

- Add the following line in place of the //TODO comment for storing code:

```
ar << m_pt << m_str;
```

- Add the following line in place of the //TODO comment for loading code:

```
ar >> m_pt >> m_str;
```

▶ To implement AutoClik's drawing code

- Use ClassView or WizardBar to jump to the **OnDraw** member function of `CAutoClickView`, and add the following line, just after the **ASSERT_VALID** call (you can replace the //TODO comment):

```
pDC->TextOut(pDoc->m_pt.x, pDoc->m_pt.y,
    pDoc->m_str);
```

The implementations of `OnLButtonDown` and `OnEditChangeText` make use of the helper function, `Refresh`.

▶ To implement the Refresh helper function

1 In ClassView, point the mouse cursor at the icon for class `CAutoClickDoc`, and click the right mouse button.

2 From the pop-up menu, choose Add Function.

3 In the Add Member Function dialog:

- In the Function Type edit box, type `void`.
- In the Function Declaration box, type: `Refresh()`
- In the Access area, select Public, and click OK.

ClassWizard adds the declaration to the Public section of the header file, creates a skeleton definition in the implementation file, and jumps you to the body of the definition, with the // TODO comment code highlighted so you can begin typing your application-specific code.

4 Implement the function with the following code:

```
UpdateAllViews(NULL);
SetModifiedFlag();
```

▶ To implement the mouse click handler

1 Switch to AutoClickView.cpp in the editor.

2 Ensure that CAutoClickView is selected in the WizardBar Object IDs list.

3 From the WizardBar Messages drop-list, double-click WM_LBUTTONDOWN.

4 Respond Yes when prompted to create a message handler.

Similar to what was described in the previous procedure, ClassWizard creates the handler, jumping you to the skeleton definition in the text editor.

5 Replace the highlighted comment with the following code:

```
CAutoClickDoc* pDoc = GetDocument();
pDoc->m_pt = point;
pDoc->Refresh();
```

▶ To implement the Change Text dialog box

1 From the Insert menu, choose Resource.

The Insert Resource dialog box appears.

2 Select Dialog and choose OK.

3 Open and pin down the dialog's properties page. (To open the properties page, choose Properties from the Edit menu.)

4 In the Dialog Properties page, type the following information:

- In the ID box, type `IDD_CHANGE_TEXT`.
- In the Caption box, type `Change Text`.

5 Add two controls to the dialog:

- A static text control labeled: Enter Text:
- An Edit control for the text

6 Open ClassWizard.

The Adding a Class dialog appears, with a message that `IDD_CHANGE_TEXT` is a new resource, and with the "Create a new class" option selected by default.

ClassWizard knows that a class hasn't been defined yet for your dialog resource, so it displays this dialog box to enable you to define one.

Note If you had created the dialog class before creating the dialog resource, you could specify the "Select an existing class option" in this dialog to connect the dialog to the existing class.

7 Click OK to create the dialog class.

The Create New Class dialog appears.

8 Under Class Information, in the Name box, type `CChangeText`.

Note the following default selections:

- **CDialog** is selected as the base class.
- IDD_CHANGE_TEXT is selected as the dialog ID.

9 Under Component Gallery, clear the "Add to Component Gallery" check box.

For more information about this option, see "Using Component Gallery" in Chapter 15 of the *Visual C++ User's Guide*.

10 Click Create.

ClassWizard creates the class and returns you to the main ClassWizard dialog.

11 Choose the Member Variables tab in ClassWizard.

12 Double-click IDC_EDIT1, and type `m_str` in the Member Variable Name box to add the member variable for the edit control.

13 Click OK twice.

▶ To add the Change Text command to AutoClik's Edit menu

1 From ResourceView, expand the Menu folder and double-click IDR_ACLICKTYPE.

The menu editor opens.

2 Click AutoClik's Edit menu.

3 Add a separator below the Paste menu item.

4 Add the following menu item text below the separator:

Change &Text...

5 Press ENTER.

The menu editor automatically names the command `ID_EDIT_CHANGETEXT`. You can view this by selecting the menu item again.

6 Type a prompt string such as

```
Change text displayed in the view.
```

You may want to close the resource editors before proceding to the next step.

▶ To implement the handler for the Change Text command

1 Switch to AutoClickDoc.cpp in the text editor.

2 In the WizardBar Object ID box, select ID_EDIT_CHANGETEXT.

3 In the Messages drop-list, double-click COMMAND, and accept the default name `OnEditChangetext`.

4 Fill in the skeleton `OnEditChangeText` handler as follows:

```
CChangeText dlg;
dlg.m_str = m_str;
if (dlg.DoModal())
{
    m_str = dlg.m_str;
    Refresh();
}
```

5 Add the following **#include** statement to AutoClikDoc.cpp:

```
#include "ChangeText.h"
```

Building and Running AutoClik Step 1

If you build and run AutoClik Step 1 now, it will run only as a stand-alone application and minimally as an automation server. You will be able to launch AutoClik from Autodriv, but if you try to access any of the methods or properties not yet implemented, Autodriv will not be able to find them.

At this point there is enough information for automation clients to create an AutoClik document, but not enough to call methods or get or set properties. You will add this functionality in Chapters 18 and 19.

CHAPTER 18

Implementing Automation Properties and Methods

By the end of Step 1, AppWizard has enabled AutoClik to work as an automation server. Also, AutoClik's basic behavior has been completely implemented, which is where most of the work is in a typical application. With the help of ClassWizard, you can easily add properties and methods to the dispatch maps.

In Step 2, you will:

- Expose the `CAutoClickDoc` member variable `m_pt` by using the Get and Set methods in AutoClik's document dispatch interface.
- Expose the `CAutoClickDoc` member variable `m_str` as a property in AutoClik's document dispatch interface.
- Add automation methods for `RefreshWindow`, `SetAllProps`, and `ShowWindow`.

In the course of doing this, you'll also learn about MFC OLE dispatch maps.

Implementing Properties of a Dispatch Interface

AutoClik's document class has two member variables, `m_pt` and `m_str`. They can be exposed to automation by using AutoClik's document dispatch interface.

There are two ways to expose member variables of an automated **CCmdTarget**-derived class.

- Directly expose the member variable as a dispatch interface property. This is analogous to declaring a member variable public in a C++ class so that objects of any other class can directly access the member variable.
- Indirectly expose the member variable by using a pair of dispatch interface Get and Set methods. This is analogous to declaring a member variable protected or private in a C++ class and declaring Get and Set member functions that other C++ objects must call to access the member variable.

When should you expose a member variable directly, as a dispatch interface property, and when should you expose it indirectly, by using dispatch interface Get and Set methods? Again, the question is analogous to: when should you declare a member variable protected or private and provide Get and Set member functions? If you do not need to monitor access to a member variable, you can expose it directly. If your application needs to know when the member variable is being accessed, you should expose it indirectly.

In the case of AutoClik, it makes the most sense to expose both `m_pt` and `m_str` indirectly by using Get and Set methods. This way, any time `m_pt` and `m_str` are updated through automation, AutoClik updates its view. For tutorial purposes, however, you handle `m_pt` and `m_str` differently. You expose `m_str` directly, whereas you expose `m_pt` indirectly by using the Get and Set methods. Both approaches to exposing the member variables are easy to do with the help of ClassWizard.

▶ To indirectly expose the m_pt member variable in the dispatch interface

1. Open ClassWizard.
2. Choose the OLE Automation tab.
3. In the Class Name box, select CAutoClickDoc, if it is not already selected.
4. Click Add Property.
5. In the Add Property dialog box, type `x` as the External Name.
6. Under Implementation, select Get/Set Methods.

   (You will use the other choice, Member Variable, for `m_str`.)
7. In the Type list box, select short.
8. Click OK.

   This returns you to the OLE Automation tab. The new OLE property, listed as `x` in the Name list, is implemented with Get and Set member functions. The Implementation box shows:

   ```
   short GetX( );
   void SetX(short nNewValue);
   ```

   The gray glyph with a "C" indicates that there is code associated with these member functions.
9. Choose the Edit Code button.
10. Implement the Get method with the following code (replace the highlighted comment and existing code):

    ```
    return (short)m_pt.x;
    ```

11 Implement the Set method with the following code:

```
m_pt.x = nNewValue;
Refresh();
```

The x and y members of a point are declared as **long** in Win32. For compatibility with versions of Windows that support only 16-bit GDI coordinates, the (**short**) type-cast truncates the LONG coordinate. This eliminates a compiler warning.

12 Repeat steps 1 through 9 for the y property, ending with:

```
short CAutoClickDoc::GetY()
{
    return (short)m_pt.y;
}

void CAutoClickDoc::SetY(short nNewValue)
{
    m_pt.y = nNewValue;
    Refresh();
}
```

(Note that you just add the implementation code: ClassWizard generated the stub handlers.) ClassWizard enables you to implement methods the same way you implement member functions.

ClassView displays the new methods. You can examine AutoClickDoc.cpp to see how ClassWizard updated the dispatch map of the document class:

```
BEGIN_DISPATCH_MAP(CAutoClickDoc, CDocument)
    //{{AFX_DISPATCH_MAP(CAutoClickDoc)
    DISP_PROPERTY_EX(CAutoClickDoc, "x", GetX, SetX, VT_I2)
    DISP_PROPERTY_EX(CAutoClickDoc, "y", GetY, SetY, VT_I2)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

You can see how the information you entered in ClassWizard is reflected in the dispatch map.

▶ To directly expose the m_str member variable in the dispatch interface

1 Open ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class Name box, choose CAutoClickDoc if it is not already selected.

4 Choose Add Property.

5 In the Add Property dialog box, in the External Name box, type `text1`.

6 Under Implementation, select Member Variable.

Whereas you exposed `m_pt` indirectly by using the Get/Set Methods option, expose `m_str` directly as a Member Variable.

7 In the Type list box, select CString.

8 Replace ClassWizard's proposed Variable Name, m_text (which was based on the External Name), with m_str.

Soon you will see how to associate the text dispatch property with the `m_str` member variable already declared in the document class.

9 Remove ClassWizard's proposed notification function name, OnTextChanged.

This step is included for instructional purposes. You could have implemented an `OnText1Changed` function by calling `Refresh( )`, just as you did for `SetX()` and `SetY()`. If you do not implement a similar `OnText1Changed` function, then you can see the different behavior when you drive AutoClik from an automation client. When the automation client updates the text, AutoClik does not automatically update its view as it does when the automation client changes the `x` or `y` values. Instead, the automation client must call the `Refresh` method to update AutoClik's view with the most recently changed text.

10 Click OK.

This returns you to the OLE Automation tab, which now displays the three properties: `text`, `x`, and `y`. The Implementation box for the `text` variable shows:

```
CString m_str
```

11 Click OK.

12 Open AutoClickDoc.h.

ClassWizard has declared the following members in the dispatch map:

```
//{{AFX_DISPATCH(CAutoClickDoc)
CString m_str;
afx_msg short GetX();
afx_msg void SetX(short nNewValue);
afx_msg short GetY();
afx_msg void SetY(short nNewValue);
//}}AFX_DISPATCH
DECLARE_DISPATCH_MAP()
```

At this point, the document header file declares `m_str` twice. The first declaration is the one you originally wrote:

```
// Attributes
public:
    CPoint m_pt;
    CString m_str;
```

The second declaration is the one ClassWizard added above in the dispatch map.

13 Remove the `CString m_str` declaration that you wrote, shown above.

14 Add the **public** keyword just after the line:

```
// Generated OLE dispatch map functions
```

This changes the declaration of the dispatch map from **protected** to **public**. This is necessary because `m_str` had already been declared as public so it could be accessed by the view.

Implementing Methods of a Dispatch Interface

You now add three methods to AutoClik's document dispatch interface:

Method	Description
`RefreshWindow`	Updates the view according to the current values of `m_pt` and `m_str`.
`SetAllProps`	Sets the `m_pt` and `m_str` member variables, and updates the view.
`ShowWindow`	Shows AutoClik's frame window, which is initially hidden when AutoClik is launched as an automation server.

The `RefreshWindow` method is the `Refresh` member function originally implemented in Step 1. Here you directly expose the `Refresh` member function, just as you directly exposed the `m_str` member variable of `CAutoClickDocument`.

▶ To directly expose the Refresh member function in the dispatch interface

1 Open ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class Name box, select CAutoClickDoc, if it is not already selected.

4 Click Add Method.

5 In the Add Method dialog box, type `RefreshWindow` in the External Name box.

This is the name that the automation client, Autodriv, uses to refer to the method, as in the following code:

```
void CAutoDrivDlg::OnRefresh()
{
    m_autoClikObject.RefreshWindow();
}
```

6 In the Internal Name box, replace the proposed "RefreshWindow" with Refresh.

`Refresh` is the name of the member function you implemented in Step 1. You do not need to make the Internal Name the same as the External Name, even though ClassWizard proposes that you do so.

7 In the Return Type box, type `void`, or select it from the drop-down list box.

8 Click OK.

This returns you to the OLE Automation tab. The new method, `RefreshWindow`, is shown in the Name list. The gray glyph with an "M" in it indicates that this is a method: The implementation box shows:

```
void Refresh( );
```

9 Choose the Edit Code button.

Because `Refresh` was selected in the OLE Automation tab, ClassWizard takes you to the implementation of `Refresh` in AutoClickDoc.cpp:

```
void CAutoClickDoc::Refresh()
{
    UpdateAllView(NULL);
    SetModifiedFlag();
}
```

However, you implemented the `Refresh` member function in Step 1; ClassWizard was not aware of that, so it implemented a second stub member function at the end of AutoClickDoc.cpp.

You will need to remove this second stub implementation. This is analogous to how you removed ClassWizard's redundant declaration of `m_str` earlier.

10 Remove ClassWizard's redundant (and empty) implementation of `Refresh` at the end of AutoClickDoc.cpp.

11 Remove the redundant declaration in the public Operations section of AutoClickDoc.h.

Leave the dispatch map entry created by ClassWizard:

```
afx_msg void Refresh( );
```

There are two more methods to implement: `SetAllProps` and `ShowWindow`.

▶ To add a method with parameters

1 Open ClassWizard.

2 Choose the OLE Automation tab.

3 Select CAutoClickDoc in the Class Name box, if it is not already selected.

4 Click Add Method.

5 Type `SetAllProps` in the External Name box. Accept ClassWizard's proposal to reuse this as the Internal Name, which is the name of the class member function.

6 In the Return Type box, select void.

7 Click in the Parameter List box to begin entering information for the first parameter of the `SetAllProps` method.

This will highlight the first blank row in the Parameter box.

- Under the Name heading, type `x` in the Name box.
- If necessary, click under the Type heading, and select short as the Type.

8 Repeat step 7 for the `y` parameter, entering it in the row under the `x` parameter.

9 Add the third parameter, `text`, selecting LPCTSTR from the Type drop-list.

10 Click OK.

This returns you to the OLE Automation tab of ClassWizard, which shows the following implementation:

```
void SetAllProps( short x, short y, LPCTSTR text);
```

11 Click Edit Code.

This takes you to the stub handler that ClassWizard created in AutoClickDoc.cpp:

```
void CAutoClickDoc::SetAllProps(short x, short y, LPCTSTR text)
{
    // TODO:  Add your dispatch handler code here
}
```

12 Implement the handler with the following code:

```
    m_pt.x = x;
    m_pt.y = y;
    m_str = text;
    Refresh();
```

Take a look at the dispatch map for the `SetAllProps` method (in AutoClickDoc.cpp):

```
BEGIN_DISPATCH_MAP(CAutoClickDoc, CDocument)
    //{{AFX_DISPATCH_MAP(CAutoClickDoc)
    ...
    DISP_FUNCTION(CAutoClickDoc, "SetAllProps", SetAllProps, VT_EMPTY,
        VTS_I2 VTS_I2 VTS_BSTR)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

The last four parameters of the **DISP_FUNCTION** entry for `SetAllProps` list the return type, **VT_EMPTY** for **void**, followed by the three parameters. You do not need to interpret the parameter types in dispatch maps; the framework interprets them at run time. But you can see that **VTS_I2** represents **short** and **VTS_BSTR** represents **LPCTSTR**.

The last method you'll implement is `ShowWindow`. You need this method because AutoClik leaves its frame window hidden when it is initially launched by the automation client. This is the default behavior implemented by AppWizard, which is appropriate for most automation servers. Typically, the automation server gives the automation client control over when the server window is shown or hidden. If you want your automation server to show its frame window right at the time it is launched by the automation client, simply remove the `RunAutomated` condition in the following if-statement provided by AppWizard in the application's `InitInstance` routine (in AutoClik.cpp):

```
BOOL CAutoClickApp::InitInstance()
{
    ...

        // Check to see if launched as OLE server
    if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
    {
        // Application was run with /Embedding or /Automation.  Don't
show the
        //  main window in this case.
        return TRUE;
    }

    ...

// The main window has been initialized, so show and update it.
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();

    return TRUE;
}
```

▶ To add the ShowWindow method

1 Open ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class Name box, select CAutoClickDoc if it is not already selected, and choose Add Method.

4 Type `ShowWindow` in the External Name box, and accept ClassWizard's proposal to reuse this as the Internal Name, which is the name of the class member function.

5 In the Return Type box, select void.

Click OK to return to the OLE Automation tab (this method has no parameters).

6 Click Edit Code.

This takes you to the stub implementation of `ShowWindow` created in AutoClickDoc.cpp by ClassWizard.

7 Implement `ShowWindow` with the following code:

```
POSITION pos = GetFirstViewPosition();
CView* pView = GetNextView(pos);
if (pView != NULL)
{
CFrameWnd* pFrameWnd = pView->GetParentFrame();
pFrameWnd->ActivateFrame(SW_SHOW);
pFrameWnd = pFrameWnd->GetParentFrame();
if (pFrameWnd != NULL)
pFrameWnd->ActivateFrame(SW_SHOW);
}
```

Adding a Stub Property

At this point, you can build AutoClik but Autodriv won't be able to load it. In Step 3, you'll implement a Position property, which exposes the document's `m_pt` member variable to a second dispatch interface. In order to load AutoClik from Autodriv to see your Step 2 code at work, you need to add the stub Position property.

▶ To add the Position property

1. Open ClassWizard.
2. Choose the OLE Automation tab.
3. In the Class Name box, select CAutoClickDoc if it is not already selected, and choose Add Property.
4. In the Add Property dialog box, type `Position` as the External Name.
5. In the type box, select LPDISPATCH.
6. Under Implementation, choose Get/Set Mehtods.
7. Choose OK.

   This returns you to the OLE Automation tab. The new OLE `Position` property is implemented with stub Get and Set member functions. The Implementation box shows:

   ```
   LPDISPATCH GetPosition();
   void SetPosition(LPDISPATCH newValue);
   ```

   At this point, you would normally choose the Edit Code button to implement the Get and Set methods. However, you'll do that in Step 3 of the tutorial. For now, this stub property makes it possible to run Autodriv and test the code you added in Step 2.
8. Exit ClassWizard and save your changes to the project files.

Build and Test AutoClik Step 2

AutoClik is mostly implemented now. You can try it out with the Autodriv application. For more information on loading and using Autodriv, see "Preview of the AutoClik Application" in Chapter 16.

▶ To test AutoClik Step 2

1. Build and then run AutoClik.exe to register the Step 2 version as the OLE server application that Autodriv will load.
2. Close AutoClik.
3. Run Autodriv and try the following things:
   - Choose the Set X and Set Y buttons in Autodriv.
   - Change the X and Y coordinates in Autodriv and choose Set All.
   - Click around in AutoClik and choose the Get All button in Autodriv.

- Change the text in the Text box of Autodriv and choose the Set Text button.

  Nothing happens—Why?

  `Set X` and `Set Y` call AutoClik's `SetX` and `SetY` methods, which call `CAutoClickDoc::Refresh`. This means that when you use Autodriv's `Set X` and `Set Y` functions, the change shows up immediately in AutoClik's window.

  In contrast, Autodriv's `Set Text` function directly accesses the `m_str` member variable of AutoClik's document. The change does not show up in AutoClik's window until you call `Refresh Display` from Autodriv, which calls AutoClik's `RefreshWindow` method, which in turn calls `CAutoClickDoc::Refresh`.

- Choose the Refresh Display button.

  The changes you entered in the Text box are displayed in AutoClik's window.

Note that the `GetPosition` and `SetPosition` methods have not been implemented. You will implement them in Step 3.

CHAPTER 19

Implementing Multiple Dispatch Interfaces

In Step 2, you added properties and methods to AutoClik's document dispatch interface, which was initially implemented by AppWizard. In Step 3, you create an entirely new **CCmdTarget**-derived class that is exposed by using a second dispatch interface.

In this step, you will:

- Use ClassWizard to create a **CCmdTarget**-derived class, named `CAutoClickPoint`, which implements a second, unnamed dispatch interface.
- Expose AutoClik's x and y coordinates by having AutoClik's document dispatch interface refer to the second Point dispatch interface.

AutoClik's second dispatch interface, implemented in class `CAutoClickPoint`, is very simple. It has two properties: `x` and `y`. It has no methods. This dispatch interface has been included solely for tutorial reasons. AutoClik already fully exposes its behavior by using the document dispatch interface implemented in Step 2. `CAutoClickPoint`'s dispatch interface is introduced to illustrate techniques for managing multiple dispatch interfaces in the same application.

The design decision to split functionality into multiple dispatch interfaces is no different from design decisions to split a C++ application into multiple classes. This basic approach is so strong that the framework enforces a one-to-one relationship between dispatch interfaces and automation-enabled **CCmdTarget**-derived classes.

AutoClik's document dispatch interface will refer to the second dispatch interface in its implementation of the `GetPosition` and `SetPosition` methods. AutoClik's document dispatch interface will expose the x and y coordinates using this Point interface as a programmatic alternative for the automation client. The automation client can get or set AutoClik's x and y coordinates by using the `GetX`, `GetY`, `SetX`, and `SetY` methods implemented in Step 2. Alternatively, the automation client can use the `GetPosition` and `SetPosition` methods implemented here in Step 3.

Creating a New CCmdTarget Class with a Dispatch Interface

When you use ClassWizard, it's simple to derive a new class from **CCmdTarget** that implements a new dispatch interface.

▶ To create a new CCmdTarget class with a dispatch interface

1 Open ClassWizard.

2 Click the Add Class menu button, and from the menu choose New.

The Create New Class dialog box appears.

3 Under Class Information, in the Name box, type `CAutoClickPoint`.

4 From the Base Class drop-box, select CCmdTarget.

Notice that the OLE Automation options are now available, because these options pertain only to **CCmdTarget**-derived classes.

5 Select the Automation option.

The Createable by Type ID option is explained later.

6 Clear the Add to Component Gallery check box.

7 Click Create.

You are returned to the ClassWizard dialog box.

8 Choose the OLE Automation tab, and ensure that CAutoClickPoint is selected in the Class Name box.

9 Click Add Property and in the Add Property dialog box:

- Type x in the External Name box.
- Select short in the Type box.
- Accept m_x as the Variable Name.
- Remove OnXChanged as the Notification Function.

  As you will see later, the members of the `CAutoClickPoint` dispatch interface class do not need notification functions.
- Use the default Implementation type, Member Variable.

10 Click OK.

11 Repeat step 9 for the y property.

12 Click OK twice.

Take a look at the `CAutoClickPoint` class created by ClassWizard in AutoClickPoint.h and AutoClickPoint.cpp.

Tip You can use ClassView to jump to the header file by double-clicking the icon for the class.

Towards the end of the file, you'll find the declaration for the dispatch map:

```
DECLARE_DISPATCH_MAP()
```

The AutoClickPoint.cpp file implements the dispatch map, reflecting the two properties you added in ClassWizard, `x` and `y`.

Tip To jump to the implementation file, from ClassView, right-click on any of the members of `CAutoClickPoint`, and choose Go to Definition.

```
BEGIN_DISPATCH_MAP(CAutoClickPoint, CCmdTarget)
    //{{AFX_DISPATCH_MAP(CAutoClickPoint)
    DISP_PROPERTY(CAutoClickPoint, "x", m_x, VT_I2)
    DISP_PROPERTY(CAutoClickPoint, "y", m_y, VT_I2)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

Referring to One Dispatch Interface from Another

You will now implement the Position property you added at the end of Step 2. This property exposes the document's `m_pt` by using the second dispatch interface implemented by `CAutoClickPoint`. You will expose this new property by adding code to the Get and Set member functions (`GetPosition` and `SetPosition`) which you declared in Step 2. The return type of `GetPosition` and the type of the parameter passed to `SetPosition` is **LPDISPATCH**, a pointer to an OLE dispatch object.

The following two Autodriv message handlers, `OnGetPosition` and `OnSetPosition`, found in the `CAutoDrivDlg` class, access the `Position` property of AutoClik's document dispatch interface using the `GetPosition` property:

```
void CAutoDrivDlg::OnGetPosition()
{
    CClikPoint point;
    point.AttachDispatch(m_autoClikObject.GetPosition());

    m_x = point.GetX();
    m_y = point.GetY();
    UpdateData(FALSE);
}

void CAutoDrivDlg::OnSetPosition()
{
    CClikPoint point;
    point.AttachDispatch(m_autoClikObject.GetPosition());
```

```
    UpdateData(TRUE);
    point.SetX((short)m_x);
    point.SetY((short)m_y);
    m_autoClikObject.SetPosition(point.m_lpDispatch);
}
```

The code

```
    CClikPoint point;
    point.AttachDispatch(m_autoClikObject.GetPosition());
```

accesses AutoClik's `Position` property, which is declared as **LPDISPATCH** in the MFC OLE Automation server's dispatch map. The automation client gets initial access to AutoClik's document dispatch interface object by creating it (in the `OnCreate` method):

```
    if (!m_autoClikObject.CreateDispatch(_T("AutoClick.Document")))
    {
        AfxMessageBox(IDP_CANNOT_CREATE_AUTOCLICK);
        return -1; // fail
    }
    m_autoClikObject.ShowWindow();
```

▶ To declare one dispatch interface object as a property of another dispatch interface

1 Open ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class Name box, choose CAutoClickDoc if it is not already selected.

4 In the External Name box, select Position.

This is the property you added in Step 2.

5 Click Edit Code to implement the Get and Set member functions.

This takes you to their stub implementations in AutoClickDoc.cpp.

6 Implement the Get and Set member functions as shown by the following code:

```
LPDISPATCH CAutoClickDoc::GetPosition()
{
    CAutoClickPoint* pPos = new CAutoClickPoint;
    pPos->m_x = (short) m_pt.x;
    pPos->m_y = (short) m_pt.y;

    LPDISPATCH lpResult = pPos->GetIDispatch(FALSE);
    return lpResult;
}

void CAutoClickDoc::SetPosition(LPDISPATCH newValue)
{
    CAutoClickPoint* pPos =
    (CAutoClickPoint*)CCmdTarget::FromIDispatch(newValue);
    if (pPos != NULL && pPos->IsKindOf
    (RUNTIME_CLASS(CAutoClickPoint)))
    {
```

```
        m_pt.x = pPos->m_x;
        m_pt.y = pPos->m_y;
        Refresh();
    }
}
```

7 Add the following **#include** statement at the top of AutoClickDoc.cpp:

```
#include "AutoClickPoint.h"
```

This is required because the implementation of `SetPosition` refers to the `CAutoClickPoint` class.

8 Change the declaration of `CAutoClickPoint`'s OLE dispatch map (in file AutoClickPoint.h) from **protected** to **public**, by adding the **public** keyword just before the comment line:

```
public:
    // Generated OLE dispatch map functions
    //{{AFX_DISPATCH(CAutoClickPoint)
    short m_x;
    short m_y;
    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
```

This is required because `CAutoClickDoc` directly accesses `CAutoClickPoint`'s member variables `m_x` and `m_y` in its implementation of `GetPosition` and `SetPosition`.

The implementation of `GetPosition` creates a new `CAutoClickPoint` object. The `CAutoClickPoint` object, which is an automation-enabled **CCmdTarget** object, in turn creates a dispatch interface object, through the help of the framework.

Finally, `GetPosition` gets the OLE **IDispatch** pointer by calling the **CCmdTarget::GetIDispatch** member function of the `CAutoClickPoint` object and returns this **IDispatch** pointer to the automation client. The **AddRef** parameter of **GetIDispatch** is **FALSE**, because the OLE reference count of this dispatch interface object was already set to 1 when the `CAutoClickPoint` object was constructed.

The implementation of `SetPosition` does a C++ down-casting of the **IDispatch** pointer to a `CAutoClickPoint` pointer. It tests the down-casting with **IsKindOf** to make sure the automation client passed back an **IDispatch** pointer to a `CAutoClickPoint` object rather than an **IDispatch** pointer to some other kind of object.

Finally, `SetPosition` updates the view to reflect the new position of the text by calling the document's `Refresh` function. Because the `Refresh` is called by `SetPosition`, it is not necessary to implement the `OnXChanged` and `OnYChanged` member functions to update the views for the `CAutoClickPoint` class.

Createable OLE Dispatch Interface Objects

Earlier you were told not to choose the OLE Createable option for `CAutoClickPoint` in ClassWizard. You did not need to do this because the **IDispatch** pointer to the **Point** object was passed between the automation client and server by using the `SetPosition` and `GetPosition` methods of AutoClik's document dispatch interface.

If you had chosen the OLE Createable option, ClassWizard would have required you to specify the External Name of the dispatch interface, such as "AutoClik.Point". In that case, an automation client could dynamically create a `CAutoClickPoint` object, using

```
CAutoClikPoint point;
point.CreateDispatch("AutoClik.Point");
```

Build and Run

Build the Step 3 version of AutoClik. It now performs exactly as you previewed it in Chapter 16. You have finished the AutoClik tutorial.

CHAPTER 20

Building an OLE Control

An "OLE control" is an OLE object with an extended interface that lets it behave like a control for Windows.

In this tutorial you will use the OLE control development tools included with Visual C++ to create a simple OLE control called Circle. The Circle control demonstrates most of the features of an OLE control, such as properties, events, property pages, and data binding.

Note This tutorial assumes that you are familiar with Microsoft Visual C++ and the basics of the Microsoft Foundation Class Library. If you are not, follow the Scribble Tutorial beginning with Chapter 2. Scribble introduces important class library concepts and techniques, and teaches you how to use ClassWizard, Visual C++, and Microsoft Developer Studio.

The Tutorial Example: Circle

The Circle tutorial consists of three steps, CIRC1, CIRC2, and CIRC3. The individual steps may be found under Samples \ MFC Samples \ Tutorials in Books Online or in the \SAMPLES subdirectory on the Visual C++ CD-ROM. Each step contains a project file, complete source files, and other related files for a version of the Circle control that corresponds to a stage of the tutorial. CIRC1 shows Circle just after it has been created. CIRC2 shows Circle after several properties and events have been added. CIRC3 shows the completed Circle control.

In successive chapters, you will learn how to:

- Create a skeleton OLE control.
- Use Test Container to test OLE.
- Change the painting behavior of an OLE control.
- Add stock properties to an OLE control.
- Add custom properties to an OLE control.
- Make an OLE control respond to mouse events.
- Add custom events to an OLE control.

- Use text and fonts in an OLE control.
- Implement OLE control property pages.
- Use simple data binding for control properties.

Note The sample source code in CIRC1, CIRC2, and CIRC3 is not exactly the same as the source code produced by the tutorial. In the tutorial, where strings, identifiers, or filenames contain the string "circ", CIRC1 uses "circ1", CIRC2 uses "circ2", and CIRC3 uses "circ3". The main control class in the tutorial is `CircCtrl`, but in the CIRC3 sample it is `Circ3Ctrl`. This naming difference permits each stage of the Circle control to be registered as a distinct OLE control. All three of the sample controls and the control you develop by following the tutorial can be used in Test Container at the same time. The sample controls behave the same way as the control produced by following the tutorial.

Other OLE Control Samples

The CIRC1, CIRC2, and CIRC3 samples are just a few of the OLE control samples included with the Visual C++. Other samples include:

- BUTTON–A control subclassed from a Windows button control. Demonstrates use of an in-place active menu, a stock property page, and the About box control option.
- DB–A control that illustrates the usage of the **CRecordSet** and **CDatabase** classes.
- LICENSED–A control that enforces use of a design-time and run-time license.
- LOCALIZE–A control with a localized user interface. Demonstrates use of separate type libraries and resource DLLs for localization.
- PAL–A control that displays the colors of a palette. Demonstrates read-only properties, persistent Get/Set properties, persistent parameterized properties, and picture properties.
- PUSH–A control subclassed from a Windows owner-drawn button control. Demonstrates stock properties, custom events, and picture holders.
- REGSVR–Registers controls in the system registry.
- SPINDIAL–A control with the visual appearance of a spin-dial. Demonstrates property page data validation.
- TIME–A control which is invisible at run time and fires a timer event at set intervals. Demonstrates notification functions and ambient properties.
- XLIST–A control, subclassed from a Windows list box, that displays text or bitmap items. Demonstrates methods, ambient properties, picture holders, and font holders.

Creating the Circle Control

The Circle sample tutorial uses the Circle control to explore the features and functions provided by the OLE control classes. Each chapter in this tutorial represents one step in the development cycle.

Note You can find the code produced by working through this part of the tutorial in the CIRC1 sample source code directory.

The Circle control created at the beginning of the tutorial is a simple one; it meets minimum control requirements and provides only default behaviors.

In the following sections, you will preview the Circle control and learn how to:

- Generate the Circle control template using ControlWizard.
- Modify the default control bitmap.
- Modify the default About Circ Control dialog box.
- Build the control.
- Register the control.
- Test the control using Test Container.

Previewing the Circle Control

Before you work through the steps for creating the Circle control, you may want to try out the completed control in Test Container. This will help you understand OLE controls and OLE control containers in general, and the Circle control in particular. The CIRC3 sample is an OLE control that is very similar to the completed Circle control.

Note The following procedures for building, registering, and testing the CIRC3 sample control assume that Microsoft Developer Studio is running.

You'll begin by building the CIRC3 sample. An OLE control is built in much the same way as a typical dynamic-link library (DLL).

▶ To build the CIRC3 version of the Circle control

1. Open the CIRC3.MDP project file.
2. From the Build menu, choose Build CIRC3.OCX, or click the Build button on the toolbar.

After successfully building the CIRC3 control, insert it into Test Container and experiment with it.

▶ To insert the CIRC3 control into Test Container

1 From the Tools menu, choose OLE Control Test Container.

2 From the Test Container Edit menu, choose Insert OLE Control.

The Insert OLE Control dialog box appears.

3 In the Object Type list box, select CIRC3 Control and click OK.

A CIRC3 control with a hashed border and resize handles is inserted into Test Container.

You can now experiment with CIRC3 to see how it works. Try any of the following:

- Click the Test Container window outside of the control. The hashed border and resize handles disappear.
- Click the control. The hashed border and resize handles reappear.
- Move the control around by clicking the hashed border and dragging the control to different places in the window.
- Change the size and shape of the control by clicking any of the resize handles and dragging the control outline to a different shape.

When you have finished, quit Test Container.

By following similar steps, you can build CIRC1 and CIRC2 and insert them into Test Container.

Creating the Basic Control

The first step in developing an OLE control is use ControlWizard to create the project. ControlWizard creates the framework of an OLE control project, including a basic set of classes, resources, and definition files. You will use ControlWizard to create the first version of the Circle control, which will draw itself as the outline of an ellipse. No properties, events, or methods are implemented in this step.

For more information on AppWizard, see Chapter 1, "Creating Applications Using AppWizard," in the *Visual C++ User's Guide*.

▶ To create the Circle control

1 From the File menu, choose New.

The New dialog box appears.

2 In the New dialog box, select Project Workspace.

3 Click OK.

The New Project Workspace dialog box appears.

4 In the Name text box, type `Circ`.

5 In the Type list, select OLE ControlWizard.

6 In the Platforms box, choose the appropriate platforms (in this case, Win32).

7 Using the Location edit box and Browse button, select an appropriate project path.

8 Click Create.

The first OLE ControlWizard dialog box appears.

9 Click Finish.

OLE ControlWizard closes and the New Project Information dialog box appears.

10 Click OK.

The New Control Information dialog box closes and the project is created and opened in Microsoft Developer Studio.

ControlWizard creates all of the necessary files to build the Circle control. Of these files, three class templates are created:

Class	Files	Comments
`CCircApp`	CIRC.H CIRC.CPP	Implements the main DLL source. Typically, there is no need to modify this code.
`CCircCtrl`	CIRCCTL.H CIRCCTL.CPP	Implements the actual control functionality. Modify this class's code to implement control-specific behavior.
`CCircPropPage`	CIRCPPG.H CIRCPPG.CPP	Provides a template for the control's property page. Modify this class and its dialog template to implement a control-specific property page.

ControlWizard creates several other files that you will modify later in this chapter and in subsequent chapters.

File	Comments
CIRL.ODL	Textually defines the control's type information. This file is modified by ClassWizard when you add properties, events, or methods to the control. MKTYPLIB.EXE uses this file as input to generate the type library (CIRC.TLB) information that is ultimately added to the control's executable file as a resource.
CIRC.RC	Standard resource file. Contains a template for the control's property page.
CIRC.RC2	Contains user-defined resources that define a control's version information, include its type library information, and state that the control is self-registering.
CIRCCTL.BMP	The tool palette representation of the control.
CIRC.ICO	The About box dialog icon.

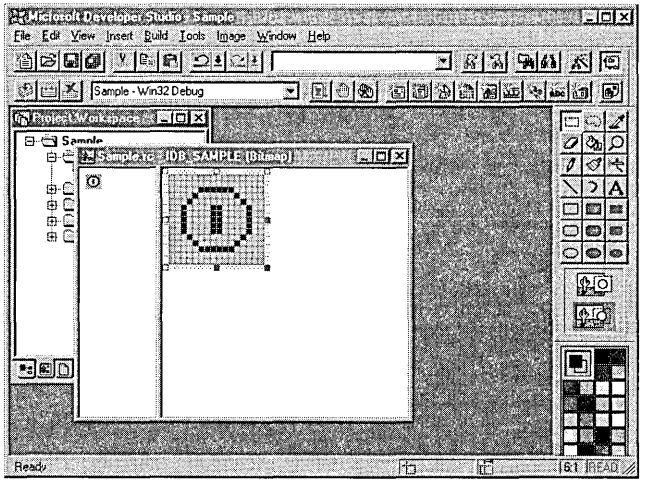
ControlWizard also creates several other standard files: CIRC.CLW, CIRC.DEF, CIRC.MAK, CIRC.VCW, MAKEFILE, README.TXT, RESOURCE.H, STDAFX.CPP, and STDAFX.H. For further infomation on files produced by ControlWizard, see the article "OLE ControlWizard: Files Created" in *Programming with MFC*.

Modifying the Control Bitmap

When ControlWizard created the Circle control classes, resources, and other files, it also generated a bitmap file called CIRCCTL.BMP. This bitmap provides a tool palette representation of the control. Tools such as Visual Basic can use this bitmap by loading it from the DLL and displaying it in a palette. The palette provides a pictorial representation of all the controls that are available to the user.

VBX controls are required to supply both an up (unselected) and down (selected) bitmap. This is not the case with OLE controls. Only one bitmap is supplied with a control, and it is up to the palette implementer to perform the appropriate processing to achieve a three-dimensional look when the control is selected and unselected in the palette.

Figure 20.1 Editing the Circle Palette Bitmap with the Bitmap Editor



▶ To modify the control bitmap

1 In the ResourceView pane, open the Circ project folder.

2 Open the Bitmap folder.

3 Double-click IDB_CIRC to start the resource editor for that bitmap.

4 Modify the bitmap as you like, giving it a look representative of a Circle control.

5 From the File menu, choose Save to save the changes to the bitmap.

6 Close the bitmap editor window.

Modifying the About Circ Control Dialog Box

ControlWizard created an About Circ Control dialog box template in the file CIRC.RC. It also defined the AboutBox "method" for the Circle control. A method is a function in a control that can be called from outside the control. The About Circ Control dialog box is displayed when the control container calls the control's AboutBox method.

ControlWizard also created an icon file called CIRC.ICO, which is displayed in the About Circ Control dialog box. You will use the resource editor to customize this icon in much the same way as you modified the control bitmap.

▶ To modify the About dialog box icon

1 In the ResourceView pane, double-click the Circ project folder.

2 Open the Icon folder.

3 Double-click IDI_ABOUTDLL to start the resource editor for that icon.

4 Edit the icon. Like the control bitmap, the icon should have a look representative of the Circle control.

5 From the File menu, choose Save to save the changes to the icon.

6 Close the icon editor window.

Building the Control

Now that you have created all required Circle control files and modified the control bitmap, you can build the control.

▶ To build the Circle control

- From the Build menu, choose Build CIRC.OCX.

Notice that the type library (TLB) compilation is automatically handled as part of the normal build process.

The build process automatically registers the Circle control. However, it is also possible to manually register a control. For more information, see the section "Registering the Control."

Registering the Control

Use the following procedure to manually register a control in the Registration Database. Note that this is automatically done as part of the build process for an OLE control.

▶ To register the Circle control

1 From the Tools menu, choose Register Control.

A message box appears, indicating that the registration was successful.

2 Click OK to close the message box.

Registering a control adds the following information to the Registration Database:

- The text name of the control
- The class name of the control
- An indicator stating that the control conforms to the OLE control protocols
- The path of the control's executable
- The path and resource ID of the control palette bitmap
- An indication of whether the control is insertable
- The **IDispatch** IDs of the control's properties and events interfaces

Controls built with the MFC OLE control classes are self-registering because two entry points, **DLLRegisterServer** and **DLLUnregisterServer**, in the control's executable are automatically added when ControlWizard creates the control's files. In the Circle control, these entry points are defined in the CIRC.CPP file. As their names imply, **DLLRegisterServer** and **DLLUnregisterServer** add to and remove from, respectively, the control's registration information in the Registration Database.

Testing the Circle Control

Once the Circle control has been built and registered, you can use Test Container to see how it behaves.

▶ To test the Circle control

1 From the Tools menu, choose OLE Control Test Container.

2 From the Edit menu, choose Insert OLE Control.

The Insert OLE Control dialog box appears.

3 From the Object Type list box, select Circ Control.

4 Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.

The Circle control will be displayed in the Test Container, as shown in Figure 20.2. Notice that the control is drawn as an ellipse within the bounding rectangle of the control.

5 Move the control within the container to observe how the control is redrawn. Resize the control to see how its ellipse is redrawn to the size of the bounding rectangle.

6 From the Edit menu, choose Invoke Methods.

The Invoke Control Method dialog box appears.

Notice that AboutBox is selected in the Name drop-down list box. This is the only method defined in the Circle control.

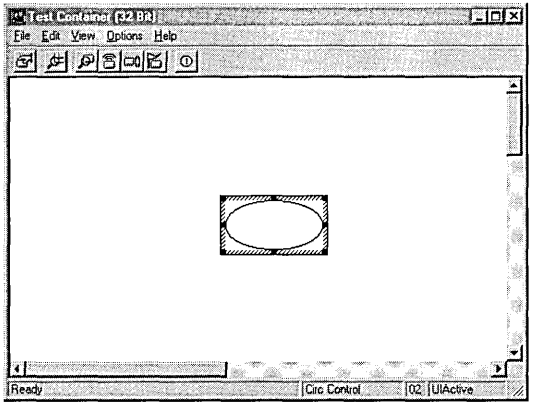
7 Click Invoke.

The About Circ Control dialog box appears. The icon you edited earlier is displayed in the dialog box.

8 Click OK to close the About Circ Control dialog box.

9 Click Close to close the Invoke Control Method dialog box.

Figure 20.2 The Circle Control



Notice that the background color of the control is white. You'll modify control painting in the next chapter.

At this point you have not defined any properties or events for the Circle control. From the Test Container View menu, choose Properties. Notice that there are no properties. From the View menu, choose Event Log. Click, move, and resize the control. No events are displayed. As the Circle control evolves, you will add a number of new properties and events.

CHAPTER 21

Painting the Control

Now that the framework for the Circle control is in place, you can modify the control to do something more useful. In this tutorial step, you'll implement the specialized background painting behavior of the Circle control.

Note You can find the code produced by working through this part of the tutorial in the CIRC2 sample source code directory.

The Circle control uses the container's ambient background color property as the default value for its background color and it uses the container's stock background color property to maintain the value of its current background color.

In this chapter, you will learn how to:

- Use the BackColor stock property.
- Set the default background color value.
- Modify the default OnDraw function to implement new painting behavior.

Enabling the BackColor Property

The first step in implementing the new painting behavior is to add the background color property (BackColor) to the control. BackColor, among others, is a stock property.

▶ To add the BackColor property

1. From the View menu, choose ClassWizard.
2. Choose the OLE Automation tab.
3. From the Class Name drop-down list box, select CCircCtrl.
4. Choose the Add Property button.

   The Add Property dialog box appears.
5. From the External Name drop-down combo box, select BackColor.
6. Under Implementation, choose Stock.

7 Click OK to close the Add Property dialog box.

This returns you to the OLE Automation tab. Notice that the implementation of the BackColor property is listed as:

```
Implementation:
Stock Property
```

8 Click OK again to confirm your choices and close ClassWizard.

ClassWizard creates the code to add the BackColor property, modifying both the `CCircCtrl` class and the type library file CIRC.ODL.

The `CCircCtrl` class's dispatch map in CIRCCTL.CPP is modified by adding the **DISP_STOCKPROP_BACKCOLOR** macro:

```
BEGIN_DISPATCH_MAP(CCircCtrl, COleControl)
    //{{AFX_DISPATCH_MAP(CCircCtrl)
    DISP_STOCKPROP_BACKCOLOR()
    //}}AFX_DISPATCH_MAP
    DISP_FUNCTION_ID(CCircCtrl, "AboutBox", DISPID_ABOUTBOX, AboutBox, VT_EMPTY,
VTS_NONE)
END_DISPATCH_MAP()
```

The control's type library file (CIRC.ODL) is modified to add BackColor to its property section, as shown in the following code sample:

```
dispinterface _DCirc
{
    properties:
        // NOTE - ClassWizard will maintain property information here.
        //    Use extreme caution when editing this section.
        //{{AFX_ODL_PROP(CCircCtrl)
        [id(DISPID_BACKCOLOR), bindable, requestedit] OLE_COLOR BackColor;
        //}}AFX_ODL_PROP
    methods:
        // NOTE - ClassWizard will maintain method information here.
        //    Use extreme caution when editing this section.
        //{{AFX_ODL_METHOD(CCircCtrl)
        //}}AFX_ODL_METHOD

        [id(DISPID_ABOUTBOX)] void AboutBox();
};
```

Because the BackColor property is of type **DISP_STOCKPROP_BACKCOLOR**, its value can be modified only through its Get and Set methods.

The value of the BackColor property is maintained by class **COleControl**. The **SetBackColor** member function of **COleControl** automatically calls the **OnBackColorChanged** member function after setting the BackColor value, invalidating the control. Invalidating the control causes the `OnDraw` function to be called, and the control is redrawn using the new background color.

Setting the Default Background Color

The next step is to provide a default value for the control's background color. Normally, the background color of a control is the same as the background color of the control container's window, which can be obtained from the container's ambient properties.

COleControl implements a mechanism for obtaining the default values of stock properties. **COleControl::OnResetState** calls the function **COleControl::DoPropExchange**, which queries the container for its background color ambient property and sets the value of the control's BackColor property equal to this color.

In projects created by ControlWizard, the default background color is implemented automatically.

Modifying the Draw Behavior

Now that you have enabled the Circle control's background color property and set its default background color, the final step is to modify the `CCircCtrl::OnDraw` function in CIRCCTL.CPP to implement the painting behavior.

As created by ControlWizard, the `CCircCtrl::OnDraw` function implements the basic Circle control drawing behavior:

```
void CCircCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    // TODO: Replace the following code with your own drawing code.
    pdc->FillRect(rcBounds, CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
    pdc->Ellipse(rcBounds);

}
```

The default behavior of the `OnDraw` function is to draw an ellipse with a white background—exactly what was displayed in Test Container earlier.

To modify `OnDraw` to use the background color value defined by the Circle control's BackColor property, remove the TODO comment line and the line on which `FillRect` is called, and modify the code as follows (beginning with the fourth line in the code example, this is new code that you will be adding):

```
void CCircCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    CBrush* pOldBrush;
    CBrush bkBrush(TranslateColor(GetBackColor()));
    CPen* pOldPen;

    // Paint the background using the BackColor property
    pdc->FillRect(rcBounds, &bkBrush);
```

```
    // Draw the ellipse using the BackColor property and a black pen
    pOldBrush = pdc->SelectObject(&bkBrush);
    pOldPen = (CPen*)pdc->SelectStockObject(BLACK_PEN);
    pdc->Ellipse(rcBounds);
    pdc->SelectObject(pOldPen);
    pdc->SelectObject(pOldBrush);
}
```

The code constructs a brush, called `bkBrush`, that uses the BackColor property color. Because a **COLORREF** value is expected for initializing the brush, and the BackColor property value is an **OLE_COLOR** value, **TranslateColor** is called first. The bounding rectangle of the control is painted using **CDC::FillRect**, specifying `bkBrush` as the fill brush.

The ellipse is drawn within the bounding rectangle of the control using the **CDC::Ellipse** member function. Before the ellipse is drawn, the background color brush and the pen must be selected into the device context. This is done by calling **CDC::SelectObject**, as shown in the code. Now when the ellipse is drawn, it is filled with the proper background color and drawn using a black pen. Finally, the old brush and pen are selected back into the device context, restoring the device context to the state in which it entered the `OnDraw` function.

Rebuilding the Control with Painting Implemented

Now that you have implemented the Circle control's painting behavior, you need to rebuild the control. Because the BackColor property was added to the control, the type library is automatically updated during the build.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the Control Drawing Behavior

To test the new drawing behavior of the control, run Test Container and load the control.

▶ To insert a Circ control in Test Container

1. Use Control Panel to modify the Windows system background color so that it is not white.
2. From the Tools menu, choose OLE Control Test Container.
3. From the Edit menu, choose Insert OLE Control.

   The Insert OLE Control dialog box appears.

4 From the Object Type list box, select Circ Control.

5 Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.

The Circle control is displayed in Test Container and painted using the same background color as the Test Container window. You can also modify the BackColor property value from the Properties dialog box in Test Container. From the View menu, choose Properties to open the Properties dialog box. Change the value of the BackColor property to 255 and click Apply. The Circle control background color changes to red.

CHAPTER 22

Adding a Custom Notification Property

The ability to define a custom set of properties, events, and methods for a control is one of the most powerful features of control writing. The previous chapter illustrated the use of a stock property. This part of the tutorial illustrates how to add a custom property to the Circle control.

Note You can find the code produced by working through this part of the tutorial in the CIRC2 sample source code directory.

OLE Control Properties–An Overview

Properties are typically used to represent control data or attributes. For example, a Date control might define a DateValue property, which would provide access to the current date value displayed in the control. This type of property represents control data. In addition, the Date control may define a Format property, which would allow the user to get and set the display format of the date. This type of property represents a control attribute. For more information about OLE control properties, see "Properties" in *Programming with MFC*.

The OLE control classes support four different types of custom properties:

- **DISP_PROPERTY**

  Implemented using a member variable.

- **DISP_PROPERTY_NOTIFY**

  Implemented using a member variable and a notification function.

- **DISP_PROPERTY_EX**

  Implemented using Get/Set functions.

- **DISP_PROPERTY_PARAM**

  Implemented using Get/Set functions and an index parameter. For more information, see "Implementing a Parameterized Property" in the article "OLE Controls: Advanced Topics."

In the following sections, you will add a property called CircleShape, which is an example of a **DISP_PROPERTY_NOTIFY** custom property. CircleShape is a Boolean property that displays the control as a perfect circle if set to **TRUE**, or as an ellipse if set to **FALSE**. (The **DISP_PROPERTY_EX** and **DISP_PROPERTY** custom property types are used later in the tutorial.)

In this chapter, you will:

- Define the CircleShape property's functionality.
- Add the CircleShape property to the control.
- Set the default value of the CircleShape property.
- Revise the control's draw behavior to reflect the value of the CircleShape property.

The CircleShape Property

This section presents general information on:

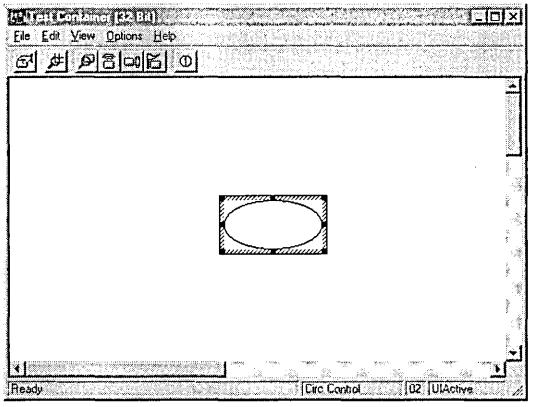
- CircleShape property functions and the property's effect on the Circle control
- Modifying the Circle control to implement CircleShape property functions

CircleShape Property Functionality

When the CircleShape property's value is set to **TRUE**, the Circle control will draw the largest possible perfect circle centered within the bounding rectangle of the control. When the CircleShape property is set to **FALSE**, the Circle control will draw an ellipse whose major and minor axes touch the bounding rectangle of the control. The initial value of the CircleShape property should be **TRUE**. Whenever the CircleShape property is changed, the Circle control should be redrawn to reflect the change.

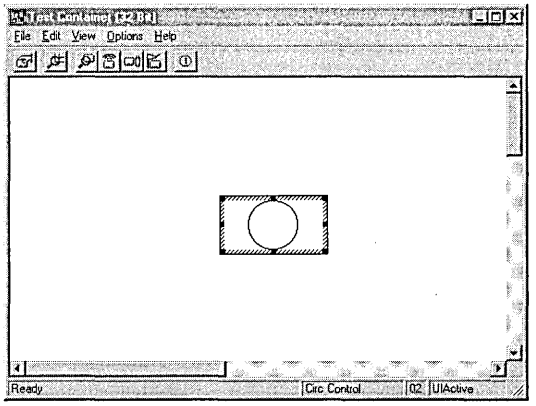
Figure 22.1 shows the Circle control drawn as an ellipse, the desired effect when CircleShape is set to **FALSE**. Notice that the ellipse is drawn to the edges of the bounding rectangle. Currently, this is the standard drawing behavior of the Circle control, so very little code needs modifying in order to implement the required drawing behavior when CircleShape is set to **FALSE**.

Figure 22.1 The CircleShape Property Set to FALSE



When the CircleShape property is set to **TRUE**, the Circle control is drawn as a perfect circle. Figure 22.2 shows how the circle would be drawn within the bounding rectangle of the control. To determine how to draw the circle, calculate the square region centered within the bounding rectangle of the control.

Figure 22.2 The CircleShape Property Set to TRUE



Recall from the previous chapter that the `CCircCtrl::OnDraw` function used the **CDC::Ellipse** function to draw the ellipse. This function can also be used to draw the circle. By passing the calculated square region instead of the bounding rectangle of the control to the **Ellipse** function, the **Ellipse** function will draw a perfect circle.

Implementing the CircleShape Property

Now that the CircleShape property's functional specification is complete, and the basic logic is described, you can revise the Circle control's code as follows:

- Add the CircleShape property to the control using ClassWizard.
- Set the default value of the CircleShape property when the control is created.
- Define the GetDrawRect member function in the `CCircCtrl` class.

  This function determines the drawing coordinates to use: if CircleShape is **FALSE**, use the entire bounding rectangle of the control; if CircleShape is **TRUE**, use the centered square region inside the bounding rectangle.
- Modify `CCircCtrl::OnDraw` to use the coordinates returned by the `CCircCtrl::GetDrawRect` member function when drawing the control.
- Modify the `CCircCtrl::OnCircleShapeChanged` member function to invalidate the control.

Aspects of this strategy apply whenever you add any custom property. Adding the property using ClassWizard greatly simplifies the process by updating the appropriate class and the object definition library (.ODL) files. It is always good practice to provide a default value for the new property by adding initialization code for the property to the `DoPropExchange` member function in the control class.

Adding the CircleShape Property

The first step in implementing the CircleShape property's functionality is to add the CircleShape property to the control. Similar to a stock property, a custom property is added using ClassWizard.

▶ To add the CircleShape property

1. From the View menu, choose ClassWizard.
2. Choose the OLE Automation tab.
3. From the Class Name drop-down list box, select CCircCtrl.
4. Click Add Property.

   The Add Property dialog box appears.
5. In the edit control of the External Name drop-down combo box, type `CircleShape`.
6. Under Implementation, choose Member Variable.
7. From the Type list box, select BOOL.
8. Verify that the Variable Name edit control contains m_circleShape.

9 Verify that the Notification Function edit control contains OnCircleShapeChanged.

10 Click OK to close the Add Property dialog box.

This returns you to the OLE Automation tab. Notice that the implementation of the CircleShape property appears as:

```
Implementation:
BOOL m_circleShape;
void OnCircleShapeChanged();
```

11 Click OK to confirm your choices and close ClassWizard.

ClassWizard creates the appropriate code to add the CircleShape property to the `CCircCtrl` class and to the CIRC.ODL file. When adding a **DISP_PROPERTY_NOTIFY** property type, ClassWizard modifies the `CCircCtrl` class's dispatch map by adding a **DISP_PROPERTY_NOTIFY** macro entry for the property:

```
BEGIN_DISPATCH_MAP(CCircCtrl, COleControl)
    //{{AFX_DISPATCH_MAP(CCircCtrl)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "CircleShape",
        m_circleShape, OnCircleShapeChanged, VT_BOOL)
    DISP_STOCKPROP_BACKCOLOR()
    //}}AFX_DISPATCH_MAP
    DISP_FUNCTION_ID(CCircCtrl, "AboutBox",
        DISPID_ABOUTBOX, AboutBox, VT_EMPTY, VTS_NONE)
END_DISPATCH_MAP()
```

The **DISP_PROPERTY_NOTIFY** macro associates the CircleShape property name with its corresponding `CCircCtrl` class member variable (`m_circleShape`), as well as the name of the `CCircCtrl` class notification function (`OnCircleShapeChanged`), which is called whenever the value of CircleShape property is changed. The property type value is specified as **VT_BOOL**.

ClassWizard also adds a declaration for the `OnCircleShapeChanged` notification function in CIRCCTL.H and a function template in CIRCCTL.CPP:

```
void CCircCtrl::OnCircleShapeChanged()
{
    // TODO: Add notification handler code

    SetModifiedFlag();
}
```

You will modify `OnCircleShapeChanged` to invalidate the control later in this chapter in "Modifying OnCircleShapeChanged."

Setting the CircleShape Default Value

Because the Circle control uses the CircleShape property as a key to determine how to draw itself, it is important to initialize the CircleShape property to a specific value. This step is easily accomplished by modifying the `CCircCtrl::DoPropExchange` function in the file CIRCCTL.CPP. Add the following line:

```
PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
```

to CIRCCTL.CPP as shown in the following code example.

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
}
```

As stated earlier in "CircleShape Property Functionality," the initialized state of CircleShape must be **TRUE** if you want the control initially drawn as a perfect circle. This is accomplished by adding a call to the **PX_Bool** function in the `DoPropExchange` function, which sets the default value of the CircleShape property to **TRUE**.

First, the property name string parameter to the **PX_Bool** function is passed through the **_T** macro. This macro is used for compatibility among different string representations. For instance, UNICODE strings are available for 32-bit OLE controls. All literal strings in an OLE control project must be handled in this manner.

Any property initialized with a call to a **PX_** function in the `DoPropExchange` function is called a "persistent property" because the **PX_** functions do more than initialize a property when a control is created. The `DoPropExchange` function is called when a control is created, restored from a file or stream, or saved to a file or stream. The **PX_** functions determine whether the value of a persistent property must change under any of these conditions. Whenever a persistent property is changed, a modified flag for persistent properties must be set to indicate that at least one persistent property must be updated. Setting the modified flag will be discussed later in this chapter, in "Modifying the OnCircleShapeChanged Function."

Implementing New Drawing Behavior

Now that you have added the CircleShape property to the `CCircCtrl` class and set its initial value in `DoPropExchange`, you can implement the new drawing behavior. To do this, you will make several additions to the `CCircCtrl` class code:

- Implement the GetDrawRect function, which calculates the coordinates of the square region.
- Modify the OnDraw function to call GetDrawRect.
- Modify the OnCircleShapeChanged function to invalidate the control.

The GetDrawRect Function

The `GetDrawRect` function determines the coordinates of the bounding rectangle in which the ellipse should be drawn. If the CircleShape property is **TRUE**, `GetDrawRect` calculates the coordinates of the square region centered in the rectangle `rc`, which was passed to `GetDrawRect`. The coordinates of the square region are put back into `rc`. If CircleShape is **FALSE**, the function leaves `rc` untouched. Add the `GetDrawRect` function at the end of the CIRCCTL.CPP file:

```
void CCircCtrl::GetDrawRect(CRect* rc)
{
    if (m_circleShape)
    {
        int cx = rc->right - rc->left;
        int cy = rc->bottom - rc->top;

        if (cx > cy)
        {
            rc->left += (cx - cy) / 2;
            rc->right = rc->left + cy;
        }
        else
        {
            rc->top += (cy - cx) / 2;
            rc->bottom = rc->top + cx;
        }
    }
}
```

Declare the `GetDrawRect` as a protected member function in the CIRCCTL.H file. Place the declaration

```
    void GetDrawRect(CRect* rc);
```

immediately after the destructor, as shown below:

```
class CCircCtrl : public COleControl
{
...
protected:
    ~CCircCtrl();
    void GetDrawRect(CRect* rc);
...
};
```

Modifying OnDraw

The `GetDrawRect` function greatly simplifies the changes that need to be made to the `OnDraw` function. The modifications to `OnDraw` introduce a local **CRect** object, `rc`, whose value is initialized to the value of `rcBounds`. Before the ellipse is drawn, `GetDrawRect` is called. If the value of the CircleShape property is **TRUE**, `GetDrawRect` adjusts the coordinates in `rc` to be the square region centered within `rcBounds`.

Add the following individual lines of code to CIRCCTL.CPP:

```
CRect rc = rcBounds;
GetDrawRect(&rc);
pdc->Ellipse(rc);
```

as shown in the following code example:

```
void CCircCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    CBrush* pOldBrush;
    CBrush bkBrush(TranslateColor(GetBackColor()));
    CPen* pOldPen;
    CRect rc = rcBounds;

    // Paint the background using the BackColor property
    pdc->FillRect(rcBounds, &bkBrush);

    // Draw the ellipse using the BackColor property and a black pen
    GetDrawRect(&rc);
    pOldBrush = pdc->SelectObject(&bkBrush);
    pOldPen = (CPen*)pdc->SelectStockObject(BLACK_PEN);
    pdc->Ellipse(rc);
    pdc->SelectObject(pOldPen);
    pdc->SelectObject(pOldBrush);
}
```

Modifying OnCircleShapeChanged

Because CircleShape is a **DISP_PROPERTY_NOTIFY** property type, ClassWizard creates the code required to force a notification function to be called if the CircleShape property's value changes. The default `OnCircleShapeChanged` notification function, created by ClassWizard, sets the modified flag for the control.

The CircleShape property is a persistent property because it is initialized by calling the **PX_Bool** function in the `COleControl::DoPropExchange` function. The modified flag for persistent properties must be set by calling the `COleControl::SetModifiedFlag` function whenever the value of the persistent property has changed. Because most properties are persistent, ClassWizard includes a call to the `CCircCtrl::SetModifiedFlag` function in all notification functions.

If the value of the CircleShape property changes, the control must redraw itself to ensure that it displays the correct representation of the control, either a circle or an ellipse. This is done by invalidating the control when the `OnCircleShapeChanged` member function is called (as shown in the following code example). The invalidation causes the `OnDraw` member function to be called.

Add the following lines of code to CIRCCTL.CPP:

```
// force the control to redraw itself
InvalidateControl();
```

as shown in the following code example:

```
void CCircCtrl::OnCircleShapeChanged()
{
    // force the control to redraw itself
    InvalidateControl();

    SetModifiedFlag();
}
```

Rebuilding the Control with CircleShape Implemented

Now that the CircleShape property is implemented, you need to rebuild the control. Because the CIRC.ODL file was changed when the CircleShape property was added to the control, the type library is automatically updated during the build.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the Control CircleShape Property

The CircleShape property's functionality has now been fully implemented. The next step is use Test Container to test the code.

▶ To insert the Circ control in Test Container

1. From the Tools menu, choose OLE Control Test Container.
2. From the Edit menu, choose Insert OLE Control.

   The Insert OLE Control dialog box appears.
3. From the Object Type list box, select Circ Control.
4. Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.

The Circle control is displayed in Test Container. Notice that the control is initially drawn as a perfect circle because the CircleShape property is set to **TRUE** in the `DoPropExchange` member function in the `CCircCtrl` class.

Next, change the value of the CircleShape property to cause the Circle control to be redrawn as an ellipse.

▶ To change the CircleShape property

1 From the View menu, choose Properties.

The Properties dialog box appears.

2 From the Property drop-down combo box, select CircleShape .

The Value edit control will display -1, which indicates a **TRUE** value. If the CircleShape property is not listed, the control might not be selected or the type library may not have been regenerated before the control was built last.

3 In the Value edit control, type 0 (zero), which indicates a **FALSE** value.

4 Click Apply.

The Circle control is redrawn as an ellipse the size of the control's bounding rectangle. It is immediately redrawn, because changing the CircleShape property caused the `OnCircleShapeChanged` notification function to be called, which invalidated the control.

CHAPTER 23

Adding a Custom Get/Set Property

The OLE control classes support a property type that can be accessed only through a Get/Set method pair. By wrapping Get and Set methods around the property, you can shield the internal representation and implementation of the property from the user.

Note You can find the code produced by working through this part of the tutorial in the CIRC2 sample source code directory.

Forcing access to a property through methods can be quite beneficial. For example, a Set method can be coded to validate an input value before the property's value is set to it, or a property can represent a computed value. When the computed property is accessed, its Get method performs a computation and returns the result as the property value.

In this chapter, you will implement a Get/Set property, CircleOffset, for the Circle control. The CircleOffset property allows the circle to be offset from the center of the control's bounding rectangle. Before the offset is modified, the Set method makes sure that the edge of the circle will stay within the control's bounding rectangle. In C++, Get and Set methods are implemented as member functions.

In this chapter, you will:

- Define the CircleOffset property's functionality.
- Add the CircleOffset property to the control.
- Set the CircleOffset property's default value.
- Set the CircleOffset property value.
- Revise the control's draw behavior to reflect the value of the CircleOffset property.
- Modify the OnCircleShapeChanged function.
- Add the OnSize function.

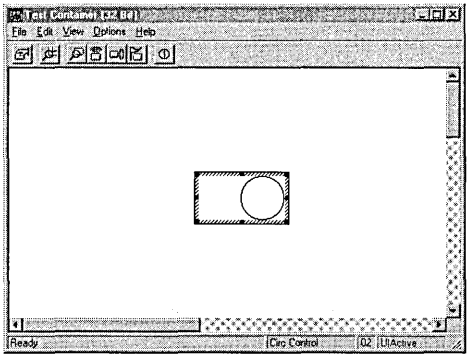
The CircleOffset Property

This section discusses the functional aspects of the CircleOffset property and the property's effect on the behavior of the Circle control. It also presents a strategy for modifying the Circle control's code to prepare for implementing the CircleOffset property's behavior.

CircleOffset Property Functionality

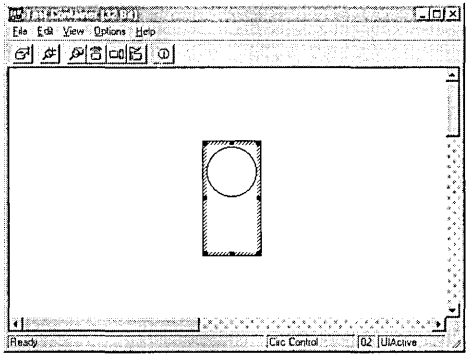
The CircleOffset property allows the user to offset the circle from the center of the control's bounding rectangle. The CircleOffset property has an effect only if the value of the CircleShape property is **TRUE**. When the CircleShape property is **FALSE**, the control is drawn as an ellipse the size of the bounding rectangle, so no movement would be possible. When CircleShape is **TRUE**, the circle can potentially be offset from center in either the x or y direction (Figure 23.1), depending on which has the greater extent.

Figure 23.1 Circle Offset 25 Units From Center



The circle can also be offset in the negative direction. For example, if the Circle control is drawn with a greater x-extent, a positive offset will move the circle to the right, and a negative offset will move the circle to the left, within the bounding rectangle. If the control is drawn with a greater y-extent, as in Figure 23.2, a positive offset will move the circle toward the top, and a negative offset will move the circle toward the bottom, of the bounding rectangle.

Figure 23.2 Circle Control With a Greater Y-Extent



The following rules apply to the behavior of CircleOffset:

- The control ignores any offset that would cause the circle to be placed outside the control's bounding rectangle.
- If the control is resized, CircleOffset is set to 0.
- If the CircleShape property is set to **TRUE**, CircleOffset is set to 0.

The control cannot allow the circle to be moved beyond its bounding rectangle. Thus, CircleOffset property's Set method must implement code that will validate the value of the offset passed to the Set method before setting the value of the CircleOffset property.

Implementing the CircleOffset Property

The implementation of the CircleOffset property's functionality is more complex than that of the CircleShape property. Not only does the CircleOffset property affect the drawing behavior of the control, it also requires that the value of the CircleOffset property be reset to 0 when either the control is resized or the CircleShape property is set to **TRUE**. Given these factors, the following code revisions must be made:

- Add the CircleOffset property to the control using ClassWizard.
- Set the default value of the CircleOffset property to 0 when the control is created.
- Modify the CircleOffset property's Set method to perform offset validation.
- Modify the `CCircCtrl::GetDrawRect` member function to properly calculate the coordinates of the square region based on the current CircleOffset property value.
- Modify the `CCircCtrl::OnCircleShapeChanged` member function to reset the CircleOffset property value to 0 if the CircleShape property is changed to **TRUE**.
- Add the `CCircCtrl::OnSize` notification function to reset the CircleOffset value to 0 if the size of the control is changed.

Adding the CircleOffset Property

The first step in implementing the CircleOffset functionality is to add the CircleOffset property to the control.

▶ To add the CircleOffset property

1 From the View menu, choose ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class Name drop-down list box, select CCircCtrl.

4 Click Add Property.

The Add Property dialog box appears.

5 In the edit control of the External Name drop-down combo box, type `CircleOffset`.

6 Under Implementation, choose Get/Set Methods.

The Get Function and Set Function edit controls appear, replacing the Variable Name and Notification Function edit controls.

7 From the Type list box, select short.

8 Click OK to close the Add Property dialog box.

This returns you to the OLE Automation tab. Notice that the implementation of the CircleOffset property appears as:

```
Implementation:
short GetCircleOffset();
void SetCircleOffset(short nNewValue);
```

9 Click OK to confirm your choices and close ClassWizard.

ClassWizard creates the appropriate code to add the CircleOffset property to the `CCircCtrl` class and to the CIRC.ODL file. Because CircleOffset is a Get/Set property type, ClassWizard modifies the `CCircCtrl` class's dispatch map to include a **DISP_PROPERTY_EX** macro entry:

```
BEGIN_DISPATCH_MAP(CCircCtrl, COleControl)
    //{{AFX_DISPATCH_MAP(CCircCtrl)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "CircleShape",
        m_circleShape, OnCircleShapeChanged, VT_BOOL)
    DISP_PROPERTY_EX(CCircCtrl, "CircleOffset",
        GetCircleOffset, SetCircleOffset, VT_I2)
    DISP_STOCKPROP_BACKCOLOR()
    //}}AFX_DISPATCH_MAP
    DISP_FUNCTION_ID(CCircCtrl, "AboutBox",
        DISPID_ABOUTBOX, AboutBox, VT_EMPTY, VTS_NONE)
END_DISPATCH_MAP()
```

The **DISP_PROPERTY_EX** macro associates the CircleOffset property name with its corresponding `CCircCtrl` class's Get and Set methods, `GetCircleOffset` and `SetCircleOffset`. The type of the property value is also specified **VT_I2**, which corresponds to **short**.

ClassWizard also adds a declaration for the `GetCircleOffset` and `SetCircleOffset` functions in CIRCCTL.H and their function templates in CIRCCTL.CPP:

```
short CCircCtrl::GetCircleOffset()
{
    // TODO: Add your property handler here

    return 0;
}

void CCircCtrl::SetCircleOffset(short nNewValue)
{
    // TODO: Add your property handler here

    SetModifiedFlag();
}
```

You will modify the `SetCircleOffset` function to perform offset validation later in this chapter.

Because ClassWizard only creates the Get and Set functions, you must add a member variable to the `CCircCtrl` class to keep track of the actual value of the CircleOffset property. The Get and Set methods will query and modify this variable. You can add this variable by modifying the declaration of the `CCircCtrl` class in the file CIRCCTL.H. An easy way to do this is to add the member variable in the protected section after the destructor.

Add the following line of code in CIRCCTL.H:

```
    short m_circleOffset;
```

as shown in the following code example:

```
class CCircCtrl : public COleControl
{
...
protected:
    ~CCircCtrl();
    void GetDrawRect(CRect* rc);
    short m_circleOffset;
...
};
```

Modify the Get method created by ClassWizard, `GetCircleOffset`, in CIRCCTL.CPP to return the value of this new variable (remove the TODO comment line):

```
short CCircCtrl::GetCircleOffset()
{
    return m_circleOffset;
}
```

Setting the CircleOffset Default Value

Similar to other properties, the CircleOffset property must be initialized to a default value when the control is created. In this case, setting CircleOffset to 0 is logical. Add the following line:

```
PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
```

to the `DoPropExchange` function in CIRCCTL.CPP:

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);

    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
    PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
}
```

The CircleOffset property is initialized by calling **PX_Short** in the `CCircCtrl::DoPropExchange` member function, passing 0 as the default value parameter (the last parameter in the function call). Now, when the circle is first drawn, it will be centered in the bounding rectangle of the Circle control. As with all literal strings, the property name parameter must be passed through the **_T** macro.

Properties initialized by calling a **PX_** function in a control's `DoPropExchange` function are persistent properties; thus, CircleOffset is a persistent property. Steps described in "Setting the Circle Offset Property" will set the modified flag for persistent properties whenever the value of the CircleOffset property changes.

Setting the CircleOffset Property

The reason that CircleOffset is defined as a Get/Set property is to provide an entry point where the offset can be validated when the user attempts to change its value. Two rules govern the offset validation:

- The CircleShape property must be set to **TRUE**.
- The new offset must not force the circle outside the control's bounding rectangle.

The Set function for Get/Set properties can be written to ignore requests to set the property value to its current value. Properties implemented using only member variables do not provide this ability. Consequently, there is another optional rule that governs validation of Get/Set properties in general:

- The new property value must be different from the old property value.

If any of the rules are not followed, `SetCircleOffset` will simply ignore the request.

If the new offset is valid, the value of the CircleOffset property is updated. Because the CircleOffset property is persistent, the modified flag is set. Because CircleOffset affects the visual appearance of the control, the control is invalidated. Invalidating the control is the simplest way to force the control to be redrawn.

Modify `SetCircleOffset` in CIRCCTL.CPP to validate the new value (the TODO comment line is removed and the `SetModifiedFlag` function call is now inside the `if` statement):

```
void CCircCtrl::SetCircleOffset(short nNewValue)
{
    // Validate the specified offset value
    if ((m_circleOffset != nNewValue) && m_circleShape && InBounds(nNewValue))
    {
        m_circleOffset = nNewValue;
        SetModifiedFlag();
        InvalidateControl();
    }
}
```

Notice that the function `SetCircleOffset` calls the `InBounds` member function. This function returns **TRUE** if the specified offset does not force the circle outside the bounding rectangle of the control.

▶ To implement the InBounds member function

1 Add the following line to your CIRCCTL.H file:

```
BOOL InBounds(short nOffset);
```

as shown in the following code example:

```
class CCircCtrl : public COleControl
{
...
protected:
    ~CCircCtrl();
    void GetDrawRect(CRect* rc);
    short m_circleOffset;
   BOOL InBounds(short nOffset);
...
};
```

2 Add the function implementation at the end of your CIRCCTL.CPP file:

```
BOOL CCircCtrl::InBounds(short nOffset)
{
    CRect rc;
    int diameter;
    int length;

    GetClientRect(rc);

    int cx = rc.right - rc.left;
    int cy = rc.bottom - rc.top;

    if (cx > cy)
    {
        length = cx;
        diameter = cy;
    }
    else
    {
        length = cy;
        diameter = cx;
    }
    if (nOffset < 0)
        nOffset = -nOffset;
    return (diameter / 2 + nOffset) <= (length / 2);
}
```

Drawing the Control

The CircleOffset functionality does not require any major drawing modifications. In fact, the `OnDraw` function need not be changed at all. The only concern is that the coordinates of the square region returned by `GetDrawRect` are properly adjusted based on the current value of the CircleOffset property:

```
void CCircCtrl::GetDrawRect(CRect* rc)
{
    if (m_circleShape)
    {
        int cx = rc->right - rc->left;
        int cy = rc->bottom - rc->top;

        if (cx > cy)
        {
            rc->left += (cx - cy) / 2;
            rc->right = rc->left + cy;

            // offset circle in bounding rect
            rc->left += m_circleOffset;
            rc->right += m_circleOffset;
        }
        else
        {
            rc->top += (cy - cx) / 2;
            rc->bottom = rc->top + cx;

            // offset circle in bounding rect
            rc->bottom -= m_circleOffset;
            rc->top -= m_circleOffset;
        }
    }
}
```

The changes to the code for `GetDrawRect` are adjustments to the square region's left and right coordinates (if the x-extent is greater) or the top and bottom coordinates (if the y-extent is greater).

Modifying the OnCircleShapeChanged Function

The CircleOffset functional specification requires that the value of the CircleOffset property be reset to 0 when CircleShape is set to **TRUE**. The `OnCircleShapeChanged` function is called whenever the CircleShape property is modified, so it is the appropriate place to handle this requirement:

```
void CCircCtrl::OnCircleShapeChanged()
{
    // force the control to redraw itself
    InvalidateControl();

    SetModifiedFlag();

    // reset the circle offset, if necessary
    if (m_circleShape)
        SetCircleOffset(0);
}
```

Setting the CircleOffset property to 0 under this condition causes the circle to be centered whenever the CircleShape property is set back to **TRUE**. All actions related to the CircleShape property change are completed before the CircleOffset property is changed.

Adding the OnSize Function

The final requirement is to set CircleOffset to 0 whenever the size of the control changes. This can be done by adding a notification function, `OnSize`, to `CCircCtrl` that responds to a **WM_SIZE** message.

▶ To add the OnSize function

1. From the View menu, choose ClassWizard.
2. Choose the Message Maps tab.
3. From the Class Name drop-down list box, select CCircCtrl.
4. From the Object IDs list box, select CCircCtrl.
5. From the Messages list box, select WM_SIZE.
6. Click Add Function.

   This returns you to the Message Maps tab. Notice that this new handler appears in the Member Functions list box as:

   ```
   Member Functions
   OnSize                    ON_WM_SIZE
   ```

7. Click Edit Code.

   ClassWizard closes and the insertion point is positioned at the code for the `OnSize` member function.

The `OnSize` member function resets the CircleOffset property to 0 only if CircleShape is **TRUE**. This ensures that the circle always stays within the bounding rectangle of the control no matter how the control is resized. Setting the CircleOffset property to 0 is the simplest way to accomplish this task.

```
void CCircCtrl::OnSize(UINT nType, int cx, int cy)
{
    COleControl::OnSize(nType, cx, cy);

    // If circle shape is true, reset the offset when control size is changed
    if (m_circleShape)
        SetCircleOffset(0);
}
```

Rebuilding the Control with CircleOffset Implemented

Now that the CircleOffset property modifications are complete, you need to rebuild the control. Because the CIRC.ODL file changed when the CircleOffset property was added to the control, the type library is automatically updated during the build.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the Control CircleOffset Property

CircleOffset functionality has been implemented and you can use Test Container to test it.

▶ To insert a Circ control in Test Container

1. From the Tools Menu, choose OLE Control Test Container.
2. From the Edit menu, select Insert OLE Control.

   The Insert OLE Control dialog box appears.
3. From the Object Type list box, select Circ Control.
4. Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.

The Circle control is displayed in Test Container's window. Notice that the control is initially drawn as a perfect circle. The default bounding rectangle of the control is a rectangle with an x-extent greater than its y-extent.

Next, change the value of the CircleOffset property to cause the circle to move from the center of the bounding rectangle:

▶ To change the CircleOffset property

1. From the View menu, choose Properties.

   The Properties dialog box appears.
2. From the Property drop-down combo box, select CircleOffset.

   The Value edit control displays 0 (zero), which is the CircleOffset property's default value.

   In the Value edit control, type 10, which indicates 10 units to the right of center. (If the y-extent of the control were larger, it would indicate 10 units to the top of center.)
3. Click Apply.

   The Circle control is redrawn, with the circle offset from center by 10 units.

Experiment with different values for the CircleOffset property: type a negative number to force the circle to the left of center. Try a number larger than the width or height of the control; for example, 2000. Notice that nothing happens. In the Properties dialog box, the number in the Value edit control reverts to CircleOffset's last valid value.

Now, resize the control so that its y-extent is greater than its x-extent. Specify positive and negative values for the CircleOffset property to see how the circle moves above and below the center of the control's bounding rectangle.

CHAPTER 24

Adding Special Effects

This chapter further evolves the functionality of the Circle control. A new behavior is introduced that produces special effects in response to mouse events at run time.

Note You can find the code produced by working through this part of the tutorial in the CIRC2 sample source code directory.

When you click the left mouse button inside the circle, the control briefly flashes a different color. To implement this special effect, the control must respond to mouse events.

This part of the tutorial introduces a new custom member variable property called FlashColor. The value of the FlashColor property contains the color the control will flash when the left mouse button is clicked when the insertion point (or cursor) is within the circle.

In this chapter, you will:

- Add the FlashColor property to the Circle control.
- Set the value of the default color for FlashColor.
- Implement code to respond to mouse events.
- Implement code to perform hit testing.
- Add the FlashColor function.

Adding the FlashColor Property

The FlashColor property holds the color value that flashes within the control circle. Windows represents a color as a 32-bit value, defined as a **COLORREF** type. The OLE control classes do not directly support the **COLORREF** type, but they do support an **OLE_COLOR** type that can hold the required information. Thus, the FlashColor property is defined as an **OLE_COLOR** type.

The FlashColor property requires no special processing when its value is changed or accessed. For this reason, FlashColor can be defined as a simple member variable property.

▶ To add the FlashColor property

1 From the View menu, choose ClassWizard.

2 Choose the OLE Automation tab.

3 From the Class Name drop-down list box, select CCircCtrl.

4 Click Add Property.

The Add Property dialog box appears.

5 In the the edit control of the External name drop-down combo box, type `FlashColor`.

6 From the Type list box, select OLE_COLOR.

7 Under Implementation, choose Member Variable (it may already be selected).

8 Verify that the Notification function edit control contains OnFlashColorChanged.

9 Click OK to close the Add Property dialog box.

This returns you to the OLE Automation tab. Notice that the implementation of the FlashColor property appears as:

```
Implementation
OLE_COLOR m_flashColor;
void OnFlashColorChanged();
```

10 Click OK to confirm your choices and close ClassWizard.

ClassWizard creates the code to add the FlashColor property, modifyng both the `CCircCtrl` class and the CIRC.ODL file. Because FlashColor is a member variable property type, ClassWizard modifies the `CCircCtrl` class's dispatch map in CIRCCTL.CPP to include a **DISP_PROPERTY_NOTIFY** macro entry:

```
BEGIN_DISPATCH_MAP(CCircCtrl, COleControl)
    //{{AFX_DISPATCH_MAP(CCircCtrl)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "FlashColor",
        m_flashColor, OnFlashColorChanged, VT_COLOR)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "CircleShape",
        m_circleShape, OnCircleShapeChanged, VT_BOOL)
    DISP_PROPERTY_EX(CCircCtrl, "CircleOffset",
       GetCircleOffset, SetCircleOffset, VT_I2)
    DISP_STOCKPROP_BACKCOLOR()
    //}}AFX_DISPATCH_MAP
    DISP_FUNCTION_ID(CCircCtrl, "AboutBox",
        DISPID_ABOUTBOX, AboutBox, VT_EMPTY, VTS_NONE)
END_DISPATCH_MAP()
```

The **DISP_PROPERTY_NOTIFY** macro associates the FlashColor property name with the following:

- Its corresponding `CCircCtrl` class member variable
- The name of the CCircCtrl class notification function (`OnFlashColorChanged`) that is called whenever the value of the FlashColor property is changed
- Its type, **VT_COLOR**, which corresponds to an **OLE_COLOR** value

ClassWizard also adds a declaration for the `OnFlashColorChanged` notification function in CIRCCTL.H and a function template in CIRCCTL.CPP:

```
void CCircCtrl::OnFlashColorChanged()
{
    // TODO: Add notification handler code
    SetModifiedFlag();
}
```

Setting the Default FlashColor Value

Next, you will write the code that sets the default value of the FlashColor property. Choosing the default value is relatively unimportant for the example; any arbitrary value will do, as long as it's not the same as the background color.

Windows uses a 32-bit unsigned integer value to represent a color. The lowest three bytes specify red, green, and blue values, each with a range from 0 through 255. Therefore, the value 0x000000FF is red, 0x0000FF00 is green, and 0x00FF0000 is blue. To simplify the process of assigning color values, you will use the **RGB** macro. Its three parameters consist of red, green, and blue values, in that order.

The member function `DoPropExchange` initializes the `m_flashColor` member variable to a value corresponding to the color red. Because the **PX_Long** function expects a reference to a **long** and `m_flashColor` is an **unsigned long**, `m_flashColor` is cast to a **long** reference.

Add the following line to the `DoPropExchange` function in CIRCCTL.CPP:

```
PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor, RGB(0xFF, 0x00, 0x00));
```

as shown in the following code example:

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);

    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
    PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
    PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor, RGB(0xFF, 0x00, 0x00));
}
```

As with all literal strings, the property name string is passed through the **_T** macro before being passed as a parameter to the **PX_Long** function.

Responding to Mouse Events

To better understand how to implement FlashColor functionality, you need to understand the behavior the Circle control should have when the mouse is clicked inside it. When the left mouse button is pressed, the circle is painted using the color stored as the value of the FlashColor property. When the left mouse button is released, the circle is repainted using the color stored as the value of the BackColor property. Clicking in the circle causes it to flash.

To implement the flash behavior, the Circle control must handle mouse events. These events are mapped to the following Windows mouse messages:

Message	Response
WM_LBUTTONDOWN	Paints the circle with the color stored as the value of the FlashColor property.
WM_LBUTTONDBLCLK	Paints the circle with the color stored as the value of the FlashColor property.
WM_LBUTTONUP	Paints the circle with the color stored as the value of the BackColor property.

Notice that a **WM_LBUTTONDBLCLK** message is handled the same as a **WM_LBUTTONDOWN** message. If the left mouse button is double-clicked in the circle, the desired flash effect occurs.

The next step is to use ClassWizard to add a message handler for each of the three mouse messages.

▶ To add the message handlers

1 From the View menu, choose ClassWizard.

2 Choose the Message Maps tab.

3 From the Class Name drop-down list box, select CCircCtrl.

4 From the Object IDs list box, select CCircCtrl.

The list of message types appears in the Messages list box.

5 From the Messages list box, select WM_LBUTTONDOWN.

6 Click Add Function.

Notice that this new handler appears in the Member Functions list box as:

```
Member Functions
OnLButtonDown           ON_WM_LBUTTONDOWN
```

7 From the Messages list box, select WM_LBUTTONDBLCLK.

8 Click Add Function.

Notice that this new handler appears in the Member Functions list box as:

```
Member Functions
OnLButtonDblClk         ON_WM_LBUTTONDBLCLK
```

9 From the Messages list box, select WM_LBUTTONUP.

10 Click Add Function.

Notice that this new handler appears in the Member Functions list box as:

```
Member Functions
OnLButtonUp         ON_WM_LBUTTONUP
```

11 Select the OnLButtonDown ON_WM_LBUTTONDOWN entry in the Member Functions list box.

12 Click Edit Code.

ClassWizard closes and the cursor is positioned at the `CCircCtrl:OnLButtonDown` function in CIRCCTL.CPP.

The following code shows the fully implemented mouse message handlers. Note that some of this code must be entered by the user, other lines are inserted automatically by ClassWizard, and the TODO comment lines inserted by ClassWizard have been removed.

```
void CCircCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    CDC* pdc;

    // Flash the color of the control if within the ellipse.
    if (InCircle(point))
    {
        pdc = GetDC();
        FlashColor(pdc);
        ReleaseDC(pdc);
    }

    COleControl::OnLButtonDown(nFlags, point);
}

void CCircCtrl::OnLButtonDblClk(UINT nFlags, CPoint point)
{
    CDC* pdc;

    // Flash the color of the control if within the ellipse.
    if (InCircle(point))
    {
        pdc = GetDC();
        FlashColor(pdc);
        ReleaseDC(pdc);
    }

    COleControl::OnLButtonDblClk(nFlags, point);
}
```

```
void CCircCtrl::OnLButtonUp(UINT nFlags, CPoint point)
{
    // Redraw the control.
    if (InCircle(point))
        InvalidateControl();

    COleControl::OnLButtonUp(nFlags, point);
}
```

`OnLButtonDown` and `OnLButtonDblClk` implement the same code: The circle will be painted using the color stored as the value of the FlashColor property. The `OnLButtonUp` function invalidates the control, causing the circle to be redrawn with the default background color.

Notice that to implement the FlashColor property's behavior, two new functions, `InCircle` and `FlashColor`, have been introduced. These functions are described in the following sections.

Hit Testing

The rule governing the Circle control flash behavior is that the circle should flash only if the mouse is within the circular area of the control. To achieve this effect, you must implement "hit testing," which checks the coordinates of every mouse click within the control to see if they are within the circle. Hit testing is implemented in the Circle control by the `InCircle` function. `InCircle` returns **TRUE** if the given point is within the area of the circle or the ellipse.

▶ To implement InCircle

1 Add the following line to your CIRCCTL.H file:

```
BOOL InCircle(CPoint& point);
```

as shown in the following example:

```
class CCircCtrl : public COleControl
{
...
protected:
    ~CCircCtrl();
    void GetDrawRect(CRect* rc);
    short m_circleOffset;
    BOOL InBounds(short nOffset);
    BOOL InCircle(CPoint& point);
...
};
```

2 Add the function implementation at the end of your CIRCCTL.CPP file:

```
BOOL CCircCtrl::InCircle(CPoint& point)
{
    CRect rc;
    GetClientRect(rc);
    GetDrawRect(&rc);
```

```
    // Determine radii
    double a = (rc.right - rc.left) / 2;
    double b = (rc.bottom - rc.top) / 2;

    // Determine x, y
    double x = point.x - (rc.left + rc.right) / 2;
    double y = point.y - (rc.top + rc.bottom) / 2;

    // Apply ellipse formula
    return ((x * x) / (a * a) + (y * y) / (b * b) <= 1);
}
```

The function works by calculating whether the point is within the boundary of the ellipse. The `GetDrawRect` function is called to make the necessary adjustments to the bounding rectangle if the value of the CircleShape property is **TRUE**.

The variables `a` and `b` are set to the horizontal and vertical radii of the ellipse. Based on the given point, the variables `x` and `y` are translated into the coordinates that are offsets from the center of the ellipse. The last line returns the Boolean result of the calculation, using the standard formula for an ellipse. Note that this calculation is also valid for a circle because a circle is simply a special case of an ellipse.

The mouse message handlers perform hit testing by passing the point coordinates that they receive as parameters to the `InCircle` member function. If `InCircle` returns **TRUE**, the circle is painted appropriately.

Adding the FlashColor Function

The `FlashColor` function paints the circle using the color value stored as the value of the FlashColor property. The code is similar to the code in the `OnDraw` function. The difference is that the `FlashColor` function paints the circle itself using the value of the FlashColor property, rather than that of the BackColor property, to fill the background of the ellipse.

▶ To implement the FlashColor function

1 Add the following line to your CIRCCTL.H file:

```
void FlashColor(CDC* pdc);
```

as shown in the following example:

```
class CCircCtrl : public COleControl
{
...
protected:
    ~CCircCtrl();
    void GetDrawRect(CRect* rc);
    short m_circleOffset;
    BOOL InBounds(short nOffset);
```

```
    BOOL InCircle(CPoint& point);
    void FlashColor(CDC* pdc);
...
};
```

2 Add the function implementation at the end of your CIRCCTL.CPP file:

```
void CCircCtrl::FlashColor(CDC* pdc)
{
    CBrush* pOldBrush;
    CBrush flashBrush(TranslateColor(m_flashColor));
    CPen* pOldPen;
    CRect rc;

    GetClientRect(rc);
    GetDrawRect(&rc);
    pOldBrush = pdc->SelectObject(&flashBrush);
    pOldPen = (CPen*)pdc->SelectStockObject(BLACK_PEN);
    pdc->Ellipse(rc);
    pdc->SelectObject(pOldPen);
    pdc->SelectObject(pOldBrush);
}
```

The `flashBrush` variable constructs a solid brush using the value of the FlashColor property stored in `m_flashBrush`. Because `m_flashBrush` is an **OLE_COLOR** value, the **TranslateColor** function is called to convert it to a **COLORREF** value first. The code selects the brush into the device context `pdc`, making sure to retain the old brush value in `pOldBrush`. A stock black pen is also selected into the device context. The old pen value is saved in `pOldPen`. The ellipse is then drawn with the black pen and filled in with the color specified by the value of the FlashColor property. Finally, the code selects the original pen and brush back into the device context `pdc`. The solid brush created when `flashBrush` is constructed is deleted when the `flashBrush` destructor is called. This occurs when the `FlashColor` function returns and the `flashBrush` variable goes out of scope.

Rebuilding the Control with FlashColor Implemented

Now that the FlashColor property modifications are complete, you need to rebuild the control.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the FlashColor Property

FlashColor functionality has been implemented and you can test it using Test Container.

▶ To test the FlashColor property

1 From the Tools menu, choose OLE Control Test Container.

2 From the Edit menu, choose Insert OLE Control.

The Insert OLE Control dialog box appears.

3 From the Object Type list box, select Circ Control.

4 Click OK to close the Insert OLE Control dialog box and insert a control into Test Container.

The Circle control is displayed in the Test Container window. Move the mouse over the circle and click once. The circle flashes red, the default value to which the FlashColor property was initialized in the `CCircCtrl::DoPropExchange` member function.

▶ To change the FlashColor property

1 From the View menu, choose Properties.

The Properties dialog box appears.

2 From the Property drop-down combo box, select FlashColor.

The Value edit control displays 255, the default value of the FlashColor property.

3 In the Value edit control, type 0 (zero), which corresponds to the color black.

4 Click Apply.

5 Click in the circle to verify that it flashes black.

You can use these color values to test the control.

Color	Hex Value	Decimal Value
White	0x00FFFFFF	16777215
Black	0x00000000	0
Gray	0x00808080	8421504
Red	0x000000FF	255
Yellow	0x0000FFFF	65535
Green	0x0000FF00	65280
Cyan	0x00FFFF00	16776960
Blue	0x00FF0000	16711680
Magenta	0x00FF00FF	16711935

Note You can specify hex values in the Properties dialog box by including the "0x" prefix as shown in the table.

CHAPTER 25

Adding Custom Events to the Circle Control

This chapter illustrates how to add custom events to the Circle control. OLE controls use events to notify a container that something has happened to a control. Commonly, an event is caused by user interaction, such as mouse or keyboard input, a Windows interaction, or a special condition occurring in the control itself. When one of these actions occur, the control alerts the container by firing an event.

MFC supports two kinds of events: stock and custom. Stock events are those events that class **COleControl** handles automatically. Custom events require extra work by the developer, but allow a control to notify a container when an action specific to the control occurs. For more information on stock and custom events, see the article "Events" in *Programming with MFC*.

This part of the tutorial shows you how to add the ClickIn and ClickOut custom events. The ClickIn event is fired when the user clicks the left mouse button with the insertion point inside the control circle; ClickOut is fired when the user clicks the left mouse button with the insertion point outside the circle.

Note You can find the code produced by working through this part of the tutorial in the CIRC2 sample source code directory.

In this chapter, you will:

- Add the ClickIn event to the Circle control.
- Implement the code to fire the ClickIn event.
- Add the ClickOut event to the Circle control.
- Implement the code to fire the ClickOut event.

Adding the ClickIn Event

Events, like properties, are added to a control using ClassWizard. When you add a custom event, ClassWizard creates the code necessary to declare the event; however, the control developer must write additional code to fire the event.

An event can also define parameters it can pass to the control's container when it fires. Event parameters can be added using ClassWizard. In this part of the tutorial, you will define two parameters for the ClickIn event: x and y, which represent the x and y coordinates of the mouse position when the left mouse button is clicked.

▶ To add the ClickIn event

1 From the View menu, choose ClassWizard.

2 Choose the OLE Events tab.

3 From the Class name drop-down list box, select CCircCtrl.

4 Click Add Event.

The Add Event dialog box appears.

5 In the edit control of the External name drop-down combo box, type `ClickIn`.

6 Use the grid control to add a parameter, called x (type `OLE_XPOS_PIXELS`).

7 Use the grid control to add a second parameter, called y (type `OLE_YPOS_PIXELS`).

8 Click OK to close the Add Event dialog box.

This returns you to the OLE Events tab. Notice that the implementation of the ClickIn event appears as:

```
Implementation:
void FireClickIn(OLE_XPOS_PIXELS x, OLE_YPOS_PIXELS y);
```

9 Click OK to confirm your choices and close ClassWizard.

ClassWizard creates the code to add the ClickIn event, modifying both the `CCircCtrl` class files and the CIRC.ODL file.

ClassWizard modifies the `CCircCtrl` class event map in CIRCCTL.CPP to add a macro entry for the ClickIn event:

```
BEGIN_EVENT_MAP(CCircCtrl, COleControl)
    //{{AFX_EVENT_MAP(CCircCtrl)
    EVENT_CUSTOM("ClickIn", FireClickIn, VTS_XPOS_PIXELS  VTS_YPOS_PIXELS)
    //}}AFX_EVENT_MAP
END_EVENT_MAP()
```

The macro **EVENT_CUSTOM** associates the ClickIn event name with `FireClickIn`, the function that actually fires the event, and with the type definitions for the x and y parameters that ClickIn uses.

An inline function is added to the `CCircCtrl` class declaration in CIRCCTL.H which, when called, fires the ClickIn event. The `FireClickIn` function simply calls the `FireEvent` function to do its work. Event functions like `FireClickIn` are added to provide a type-safe way of firing an event.

```
class CCircCtrl : public COleControl
{
    ...
    //{{AFX_EVENT(CCircCtrl)
    void FireClickIn(OLE_XPOS_PIXELS x, OLE_YPOS_PIXELS y)
        {FireEvent(eventidClickIn,EVENT_PARAM(VTS_XPOS_PIXELS  VTS_YPOS_PIXELS),
    x, y);}
    //}}AFX_EVENT
    ...
}
```

Firing the ClickIn Event

Now that the ClickIn event has been declared, it can be fired. The control itself determines when a custom event should be fired. In this example, the ClickIn event fires when the user clicks the left mouse button with the insertion point inside the control circle.

The code that fires the ClickIn event should be triggered when a **WM_LBUTTONDOWN** message is received from Windows. The Circle control already has a message handler function defined for **WM_LBUTTONDOWN**—the `OnLButtonDown` function in CIRCCTL.CPP.

Add the following lines to change `OnLButtonDown` to fire the ClickIn event:

```
        // Fire the ClickIn event
        FireClickIn(point.x, point.y);
```

as shown in the following code example:

```
void CCircCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    CDC* pdc;

    // Flash the color of the control if within the ellipse.
    if (InCircle(point))
    {
        pdc = GetDC();
        FlashColor(pdc);
        ReleaseDC(pdc);

       // Fire the ClickIn event
        FireClickIn(point.x, point.y);
    }

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

Modifying `OnLButtonDown` to fire the event requires a call to the `FireClickIn` function. The call to `FireClickIn` is made only if the `InCircle` function returns **TRUE**, which means that the insertion point is within the circle.

Adding the ClickOut Event

When the user clicks the left mouse button and the insertion point is outside the control circle, the ClickOut event should fire. The ClickOut event is simpler than the ClickIn event in that it defines no arguments.

1 From the View menu, choose ClassWizard.

2 Choose the OLE Events tab.

3 From the Class Name drop-down list box, select CCircCtrl.

4 Click Add Event.

The Add Event dialog box appears.

5 In the edit control of the External Name drop-down combo box, type `ClickOut`.

6 Click OK to close the Add Event dialog box.

This returns you to the OLE Events tab. Notice that the implementation of the ClickOut event appears as:

```
Implementation:
void FireClickOut( );
```

7 Click OK to confirm your choices and close ClassWizard.

As with the ClickIn event, ClassWizard creates the code to add the ClickOut event, modifying the files CIRC.ODL, CIRCCTL.H, and CIRCCTL.CPP.

ClassWizard modifies the `CCircCtrl` class's event map in CIRCCTL.CPP to add a macro entry for the ClickOut event:

```
BEGIN_EVENT_MAP(CCircCtrl, COleControl)
    //{{AFX_EVENT_MAP(CCircCtrl)
    EVENT_CUSTOM("ClickIn", FireClickIn, VTS_XPOS_PIXELS  VTS_YPOS_PIXELS)
    EVENT_CUSTOM("ClickOut", FireClickOut, VTS_NONE)
    //}}AFX_EVENT_MAP
END_EVENT_MAP()
```

The macro **EVENT_CUSTOM** associates the ClickOut event name with the member function `FireClickOut` and with its argument type definition. The ClickOut argument type definition is set to **VTS_NONE**, which indicates that the ClickOut event passes no arguments.

An additional inline function is added to the `CCircCtrl` class declaration. The `FireClickOut` function provides a type-safe call to fire the event:

```
class CCircCtrl : public COleControl
{
    ...
    //{{AFX_EVENT(CCircCtrl)
    void FireClickIn(OLE_XPOS_PIXELS x, OLE_YPOS_PIXELS y)
        {FireEvent(eventidClickIn,EVENT_PARAM(VTS_XPOS_PIXELS  VTS_YPOS_PIXELS),
    x, y);}
```

```
    void FireClickOut()
        {FireEvent(eventidClickOut,EVENT_PARAM(VTS_NONE));}
    //}}AFX_EVENT
    ...
}
```

Firing the ClickOut Event

ClickOut fires under a condition opposite to the one that fires ClickIn. You must modify the `OnLButtonDown` function in CIRCCTL.CPP to call `FireClickOut` when the insertion point is not within the circle (that is, the `InCircle` function returns **FALSE**). The `FireClickOut` function fires the event.

Add the following lines of code to CRCCTL.CPP:

```
    else
        // Fire the ClickOut event
        FireClickOut();
```

as shown in the code example:

```
void CCircCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    CDC*    pdc;

    // Flash the color of the control if within the ellipse.
    if (InCircle(point))
    {
        pdc = GetDC();
        FlashColor(pdc);
        ReleaseDC(pdc);

        // Fire the ClickIn event
        FireClickIn(point.x, point.y);
    }
    else
        // Fire the ClickOut event
        FireClickOut();

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

Rebuilding the Control

Now that you have added the ClickIn and ClickOut events to the Circle control, you need to rebuild it.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the ClickIn and ClickOut Events

The ClickIn and ClickOut events have been defined, and the code has been implemented to fire the events. Now you can use Test Container to test them.

▶ To test the ClickIn and ClickOut events

1. From the Tools menu, choose OLE Control Test Container.
2. From the Edit menu, choose Insert OLE Control.

   The Insert OLE Control dialog box appears.
3. From the Object Type list box, select Circ Control.
4. Click OK to close the Insert OLE Control dialog box and insert a control into Test Container.
5. From the View menu, choose Event Log.

   The Event Log dialog box appears, which lists events sent to Test Container as they occur.
6. Position the mouse within the circle and click the left mouse button.

   A ClickIn entry is added to the Event Log list, and the x and y mouse coordinates are shown.
7. Position the mouse outside the circle and click the left mouse button.

   A ClickOut entry is added to the Event Log list. Notice that no arguments are shown because none were defined for this event.

CHAPTER 26

Handling Text and Fonts

This chapter explores the text and font support provided in the OLE control classes. You will add three stock properties, Caption, Font, and ForeColor, to implement a caption in the Circle control. With these properties, you can use any font to draw text in the control.

Note You can find the code produced by working through this part of the tutorial in the CIRC3 sample source code directory.

The Circle sample uses the stock Caption property to hold the text of the caption. The stock Font property holds the font that the Circle control uses to draw its caption. The resulting caption text is drawn using the color stored as the value of the ForeColor property.

For more information about stock properties, see the article "Properties" in *Programming with MFC*. Also see the article "OLE Controls: Using Fonts in an OLE Control," which contains information about font support in OLE controls.

In this chapter, you will:

- Add the stock Caption property.
- Add the stock Font property.
- Add the stock ForeColor property.
- Implement the Caption drawing behavior.
- Add stock color and font property pages.

Adding the Stock Caption Property

Use ClassWizard to add the stock Caption property to the Circle control.

▶ To add the stock Caption property

1 From the View menu, choose ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class name drop-down list box, select CCircCtrl.

4 Click Add Property.

The Add Property dialog box appears.

5 From the External name drop-down combo box, select Caption.

6 Under Implementation, choose Stock (it may already be selected).

7 Click OK to close the Add Property dialog box.

This returns you to the OLE Automation tab. Notice that the implementation of the Caption property appears as:

```
Implementation:
Stock Property
```

8 Click OK to confirm your choices and close ClassWizard.

ClassWizard creates the code to add the Caption property, modifying both the `CCircCtrl` class and the CIRC.ODL file.

ClassWizard modifies the `CCircCtrl` class dispatch map in CIRCCTL.CPP to add the **DISP_STOCKPROP_CAPTION** macro entry for the Caption property:

```
BEGIN_DISPATCH_MAP(CCircCtrl, COleControl)
    //{{AFX_DISPATCH_MAP(CCircCtrl)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "FlashColor", m_flashColor, OnFlashColorChanged,
VT_COLOR)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "CircleShape", m_circleShape, OnCircleShapeChanged,
VT_BOOL)
    DISP_PROPERTY_EX(CCircCtrl, "CircleOffset", GetCircleOffset, SetCircleOffset,
VT_I2)
    DISP_STOCKPROP_BACKCOLOR()
    DISP_STOCKPROP_CAPTION()
    DISP_FUNCTION_ID(CCircCtrl, "AboutBox", DISPID_ABOUTBOX, AboutBox, VT_EMPTY,
VTS_NONE)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

The **DISP_STOCKPROP_CAPTION** macro enables the stock Caption property in the Circle control, implemented as a Get/Set property. Note that the Caption property is an alias for the stock Text property. The Get and Set methods and notification function for the Caption property are the **GetText**, **SetText**, and **OnTextChanged** functions in class **COleControl**.

The **OnTextChanged** function is invoked when the Caption property is modified through the **SetText** function. By default, **OnTextChanged** simply invalidates the control.

Stock property notification functions such as **OnTextChanged** can be overridden in the descendant control class. For example, you may wish to provide an optimal solution for redrawing the caption other than invalidating the control, which forces the whole control to repaint.

In addition to the **GetText** function, the Caption property can be accessed directly through the **InternalGetText** function of **COleControl**. **InternalGetText** is useful when the control developer accesses the Caption property internally, for example, from the control's `OnDraw` function. The `CCircCtrl::OnDraw` function will be changed to use the **InternalGetText** function to draw the caption text.

Adding the Stock Font Property

The Circle control uses the font information contained in the stock Font property to draw the caption text. The Font property is actually a pointer to a font object that is encapsulated by class **CFontHolder**. The font object contains several properties that describe the current font. These properties are accessible through the font object's **IDispatch** interface.

▶ To add the stock Font property

1. From the View menu, choose ClassWizard.
2. Choose the OLE Automation tab.
3. In the Class name drop-down list box, select CCircCtrl.
4. Click Add Property.

   The Add Property dialog box appears.
5. From the External name drop-down combo box, select Font.
6. Under Implementation, choose Stock (it may already be selected).
7. Click OK to close the Add Property dialog box.

   This returns you to the OLE Automation tab. Notice that the implementation of the Font property appears as:

   ```
   Implementation:
   Stock Property
   ```
8. Click OK to confirm your choices and close ClassWizard.

ClassWizard creates the code to add the Font property, modifying both the `CCircCtrl` class and the CIRC.ODL file.

ClassWizard modifies the `CCircCtrl` class dispatch map to add a **DISP_STOCKPROP_FONT** macro entry for the Font property:

```
BEGIN_DISPATCH_MAP(CCircCtrl, COleControl)
    //{{AFX_DISPATCH_MAP(CCircCtrl)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "FlashColor", m_flashColor, OnFlashColorChanged,
VT_COLOR)
    DISP_PROPERTY_NOTIFY(CCircCtrl, "CircleShape", m_circleShape, OnCircleShapeChanged,
VT_BOOL)
    DISP_PROPERTY_EX(CCircCtrl, "CircleOffset", GetCircleOffset, SetCircleOffset,
VT_I2)
    DISP_STOCKPROP_BACKCOLOR()
    DISP_STOCKPROP_CAPTION()
```

```
    DISP_STOCKPROP_FONT()
    //}}AFX_DISPATCH_MAP
    DISP_FUNCTION_ID(CCircCtrl, "AboutBox",
        DISPID_ABOUTBOX, AboutBox, VT_EMPTY, VTS_NONE)
END_DISPATCH_MAP()
```

Similar to the Caption property, the Font property is implemented as a Get/Set property. The Font property supports a notification function called **OnFontChanged**, which is defined in **COleControl**. By default, this function invalidates the control. **OnFontChanged** can be overridden in the control class to provide an optimal solution to reflecting the font change.

Adding the Stock ForeColor Property

The stock ForeColor property contains the foreground color that the control uses to paint the caption text. Like the stock BackColor property, the ForeColor property is a Get/Set property type.

▶ To add the stock ForeColor property

1. From the View menu, choose ClassWizard.
2. Choose the OLE Automation tab.
3. In the Class name drop-down list box, select CCircCtrl.
4. Click Add Property.

   The Add Property dialog box appears.
5. From the External name drop-down combo box, select ForeColor.
6. Under Implementation, choose Stock (it may already be selected).
7. Click OK to close the Add Property dialog box.

   This returns you to the OLE Automation tab. Notice that the implementation of the ForeColor property appears as:

   ```
   Implementation:
   Stock Property
   ```
8. Click OK to confirm your choices and close ClassWizard.

ClassWizard adds the code to create the ForeColor property to both the `CCircCtrl` class and the CIRC.ODL file.

Implementing Caption Drawing Behavior

Now that all the required properties are in place, you can implement caption drawing. To draw the caption text, you must modify the `CCircCtrl::OnDraw` function in CIRCCTL.CPP.

The drawing code changes require that the device context passed to the OnDraw function be modified to reflect the background and foreground colors, stored as values of the BackColor and ForeColor properties, respectively. The font contained in the Font property must also be selected into the device context before the caption text can be drawn. The **InternalGetText** function retrieves the caption text drawn using the **ExtTextOut** function.

▶ To implement caption drawing in the Circle control

- Modify the OnDraw member function in CIRCCTL.CPP by adding the two following code blocks

Block 1:

```
    CFont* pOldFont;
    TEXTMETRIC tm;
    const CString& strCaption = InternalGetText();

    // Set the ForeColor property color and transparent background
mode into the device context
    pdc->SetTextColor(TranslateColor(GetForeColor()));
    pdc->SetBkMode(TRANSPARENT);
```

Block 2:

```
    // Draw the caption using the stock Font and ForeColor properties
    pOldFont = SelectStockFont(pdc);
    pdc->GetTextMetrics(&tm);
    pdc->SetTextAlign(TA_CENTER | TA_TOP);
    pdc->ExtTextOut((rc.left + rc.right) / 2, (rc.top + rc.bottom -
tm.tmHeight) / 2,
        ETO_CLIPPED, rc, strCaption, strCaption.GetLength(), NULL);
    pdc->SelectObject(pOldFont);
```

as show in the following example:

```
void CCircCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    CBrush* pOldBrush;
    CBrush bkBrush(TranslateColor(GetBackColor()));
    CPen* pOldPen;
    CRect rc = rcBounds;
    CFont* pOldFont;
    TEXTMETRIC tm;
    const CString& strCaption = InternalGetText();

    // Set the ForeColor property color and transparent background
mode into the device context
    pdc->SetTextColor(TranslateColor(GetForeColor()));
    pdc->SetBkMode(TRANSPARENT);

    // Paint the background using the BackColor property
    pdc->FillRect(rcBounds, &bkBrush);
```

```
    // Draw the ellipse
    GetDrawRect(&rc);
    pOldBrush = pdc->SelectObject(&bkBrush);
    pdc->Ellipse(rc);

    // Draw the caption using the stock Font and ForeColor properties
    pOldFont = SelectStockFont(pdc);
    pdc->GetTextMetrics(&tm);
    pdc->SetTextAlign(TA_CENTER | TA_TOP);
    pdc->ExtTextOut((rc.left + rc.right) / 2, (rc.top + rc.bottom -
tm.tmHeight) / 2,
        ETO_CLIPPED, rc, strCaption, strCaption.GetLength(), NULL);
    pdc->SelectObject(pOldFont);

    pdc->SelectObject(pOldPen);
    pdc->SelectObject(pOldBrush);
}
```

The code added in the first block in the preceding example starts by declaring three new local variables. The `pOldFont` variable holds the old font from the device context. The `tm` variable holds text metric information about the font used to draw the text. The `strCaption` variable is the text to be drawn. It is initialized by calling the **InternalGetText** function to get the value of the Caption property. The **InternalGetText** function should be used instead of the **GetText** function whenever the returned text will not be modified. This is because **GetText** returns a string of type **BSTR**, containing a copy of the caption text, that must be freed.

```
CFont pOldFont;
TEXTMETRIC tm;
const CString& strCaption = InternalGetText();
```

The text color of the device context is set to the current value of the ForeColor property. The **OLE_COLOR** value returned by the **GetForeColor** function is translated by the **TranslateColor** function into a **COLORREF** value. This value is then passed as a parameter to the **SetTextColor** function to set the text color in the device context. **OLE_COLOR** values must be translated by the **TranslateColor** function whenever a **COLORREF** value is required.

```
pdc->SetTextColor(TranslateColor(GetForeColor()));
```

The background mode of the text is made transparent in the device context by calling the **SetBkMode** function.

```
pdc->SetBkMode(TRANSPARENT);
```

In the second block of code added in the previous example, calling the **SelectStockFont** function (defined in the **COleControl** base class) selects the current font, stored as the value of the stock Font property, into the device context. The font that used to be selected into the device context is kept in the `pOldFont` variable to be selected back into the device context later.

```
pOldfont = SelectStockFont(pdc);
```

The text metrics for the font in the device context are retrieved by calling the **GetTextMetrics** function and are used to center the text vertically. The text alignment of the device context is set using the **SetTextAlign** function, and this centers the text horizontally. The text is drawn by calling the **ExtTextOut** function. The text is clipped to the bounding rectangle of the circle or ellipse (which may be different from the bounding rectangle of the control if the CircleShape property is **TRUE**) and centered both horizontally and vertically. Note that any function affecting the device context passed to the `OnDraw` function must be among the subset of functions that are allowed for both metafile device contexts and standard device contexts. For more information on the `OnDraw` function and possible modifications, see the article "OLE Controls: Painting an OLE Control" in *Programming with MFC*.

```
pdc->GetTextMetrics(&tm);
pdc->SetTextAlign(TA_CENTER | TA_TOP);
pdc->ExtTextOut((rc.left + rc.right) / 2, (rc.top + rc.bottom - tm.tmHeight) / 2,
    ETO_CLIPPED, rc, strCaption, strCaption.GetLength(), NULL);
```

The font that used to be selected in the device context is replaced using the **SelectObject** function.

```
pdc->SelectObject(pOldFont);
```

Adding the Color and Font Property Pages

MFC supports stock color and font property pages that can be easily implemented in a control by adding entries to a control's property page ID table. ClassWizard produces a default property page ID table in CIRCCTL.CPP that looks like this:

```
BEGIN_PROPPAGEIDS(CCircCtrl, 1)
    PROPPAGEID(CCircPropPage::guid)
END_PROPPAGEIDS(CCircCtrl)
```

When editing the property page section, you must modify the page count number in the **BEGIN_PROPPAGEIDS** macro to reflect the actual number of property pages implemented by the control. In the case of the Circle control, this number is 3 when the color and font property pages are added. The first property page is the default generated by ControlWizard, the second is the color property page, and the third is the font property page.

▶ To change the ID count and add lines for the color and font property pages

- Add two lines to the code generated by ClassWizard in CIRCCTL.CPP as follows:

```
BEGIN_PROPPAGEIDS(CCircCtrl, 3)
    PROPPAGEID(CCircPropPage::guid)
   PROPPAGEID(CLSID_CColorPropPage)
   PROPPAGEID(CLSID_CFontPropPage)
END_PROPPAGEIDS(CCircCtrl)
```

The default property page allows properties to be viewed and edited. MFC also provides stock property page support for picture properties. For more information on property pages, see "OLE Controls: Property Pages" in *Programming with MFC*.

Rebuilding the Control with Font and Color Support Implemented

Now that font and color support and two stock property pages have been added, you need to rebuild the control. Since the CIRC.ODL file was changed, the type library is automatically updated during the build.

▶ To rebuild the control

- From the View menu, choose Build CIRC.OCX.

Testing the Caption Property

Use Test Container to test the caption drawing behavior.

▶ To test the caption property

1. From the Tools menu, choose OLE Control Test Container.
2. From the Edit menu, choose Insert OLE Control.

   The Insert OLE Control dialog box appears.
3. From the Object Type list box, select Circ Control.
4. Click OK to close the Insert OLE Object dialog box and insert a control into Test Container.
5. From the View menu, choose Properties.

   The Properties dialog box appears.
6. From the Property drop-down combo box, select Caption.
7. In the Value edit control, type `Hello`.
8. Click Apply.

The string "Hello" is displayed in the circle. Now try typing some longer phrases for the caption. Notice that the phrase is clipped to the bounds of the circle.

Next, modify the caption color and font.

▶ To modify the color and font properties

1. From the Edit menu, choose Properties... Circ Control Object.

   The Circ Control Properties property page dialog appears.

2 Choose the Fonts tabs.

Modify the different font properties. Click Apply whenever you want to see the control reflect your changes.

3 Choose the Colors tabs.

Modify the different color properties. Click Apply whenever you want to see the control reflect your changes. Notice that the Property Name drop-down list box contains three entries—the stock BackColor, stock ForeColor, and custom FlashColor properties

CHAPTER 27

Modifying the Default Property Page

This chapter illustrates how to work with the Circle control default property page. The default property page is a dialog box that allows the user to add controls for viewing and changing properties. ControlWizard supplies a default property page when it creates the OLE control project.

Note You can find the code produced by working through this part of the tutorial in the CIRC3 sample source code directory.

A property page provides the interface for viewing and editing a control's properties at design time. OLE controls use the default property page to display and modify properties that are not handled by stock property pages, such as the stock Font property page added to the Circle control in Chapter 26, "Handling Text and Fonts." For more information about stock and custom property pages, see the article "OLE Controls: Using Stock Property Pages" in *Programming with MFC*.

This part of the Circle tutorial modifies a default property page to display and modify the Caption property.

In this chapter, you will:

- Add controls to the default property page.
- Link the controls to the Caption, CircleOffset, and CircleShape properties.

Adding Controls to the Default Property Page

When ControlWizard created the Circle control, a default property page was included as part of the project. Property pages are listed in the property page ID table in CIRCCTL.CPP:

```
BEGIN_PROPPAGEIDS(CCircCtrl, 3)
    PROPPAGEID(CCircPropPage::guid)
    PROPPAGEID(CLSID_CColorPropPage)
    PROPPAGEID(CLSID_CFontPropPage)
END_PROPPAGEIDS(CCircCtrl)
```

The first entry in the Property page ID table is the default (or general) property page. The second and third entries are the stock color and font property pages that were added in Chapter 26, "Handling Text and Fonts."

Initially, the default property page dialog box template is empty. You can use the dialog editor to add controls to the default property page.

▶ To add the Caption property to the default property page

1. In the ResourceView pane, open the Circ project folder.
2. Open the Dialog folder.
3. Open the IDD_PROPPAGE_CIRC entry in the Dialog folder to edit the property page template.
4. Click on the default static text control in the dialog template
5. From the Edit menu, choose Cut to delete the default static text control.
6. Select the Static Text tool in the Control Palette and place a static text control in the dialog.
7. Double-click the static text control to bring up the Test Properties dialog box.
8. Using the Test Properties dialog box, change the static text control caption to &Caption:.
9. Select the Edit Box tool in the Control Palette and place an edit box control next to the static text control in the dialog.
10. Double-click the edit box control to bring up the Test Properties dialog box.
11. Using the Test Properties dialog box, change the edit control's ID to IDC_CAPTION.

▶ To add the CircleOffset property to the default property page

1. Select the Static Text tool in the Control Palette and place another static text control in the dialog.
2. Double-click the static text control to bring up the Test Properties dialog box.
3. Using the Test Properties dialog box, change the static text control caption to Circle&Offset:.
4. Select the Edit Box tool in the Control Palette and place an edit box control next to the static text control in the dialog.

5 Double-click the edit box control to bring up the Test Properties dialog box.

6 Using the Test Properties dialog box, change the edit control's ID to IDC_CIRCLEOFFSET.

▶ To add the CircleShape property to the default property page

1 Select the Check Box tool in the Control Palette and place a check box control in the dialog.

2 Double-click the edit box control to bring up the Test Properties dialog box.

3 Using the Test Properties dialog box, change the check box's ID to IDC_CIRCLESHAPE and change the check box's caption to Circle&Shape; on the Styles property sheet, check the Left Text check box.

Once you have completed these procedures, arrange the controls to your liking and choose Save from the File menu to save the property page template.

Linking Controls with Properties

Now that the property page has controls for displaying and modifying properties, the controls need to be linked to the properties. You can link controls in the property page with properties by using a shortcut to the Add Member Variable dialog box in ClassWizard.

▶ To link the property page controls to properties

1 In the ResourceView pane, open the Circ project folder.

2 Open the Dialog folder.

3 Open the IDD_PROPPAGE_CIRC entry in the Dialog folder to load the property page template.

4 While holding down the CTRL key, double-click the edit box control for the Caption property.

The Add Member Variable dialog box of ClassWizard appears.

5 After the m_ that is already in the Member variable name edit control, type `caption`, so the edit control contains m_caption.

6 In the Category drop-down list box, choose Value.

7 In the Variable type drop-down list box, choose CString.

8 In the Optional OLE property name drop-down combo box, choose Caption.

9 Click OK to close the Add Member Variable dialog box.

If you were to open ClassWizard, the Member Variable tab would contain the new member variable mapping for the Caption property, as shown below:

```
Control IDs:                Type      Member
IDC_CAPTION                 CString   m_caption
IDC_CIRCLEOFFSET
IDC_CIRCLESHAPE
```

10 Repeat steps 2 through 6, double-clicking the edit box control for the CircleOffset property, typing `circleOffset` in the Member Variable Name edit control so that it contains m_circleOffset, choosing int from the Variable Type drop-down list box, and typing `CircleOffset` in the Optional OLE Property Name drop-down combo box.

If you were to open ClassWizard, the Member Variable tab would contain the new member variable mapping for the CircleOffset property, as shown below:

```
Control IDs:                Type      Member
IDC_CAPTION                 CString   m_caption
IDC_CIRCLEOFFSET            int       m_circleOffset
IDC_CIRCLESHAPE
```

11 Repeat steps 2 through 7 double-clicking the check box for the CircleShape property, typing `circleShape` in the Member Variable Name edit control so that it contains m_circleShape, choosing BOOL from the Variable Type drop-down list box, and typing `CircleShape` in the Optional OLE Property Name drop-down combo box.

If you were to open ClassWizard, the Member Variable tab would contain the new member variable mapping for the CircleShape property, as shown below:

```
Control IDs:                Type      Member
IDC_CAPTION                 CString   m_caption
IDC_CIRCLEOFFSET            int       m_circleOffset
IDC_CIRCLESHAPE             BOOL      m_circleShape
```

ClassWizard adds the member variables to the `CCircPropPage` class. ClassWizard also adds functions to the `CCircPropPage` class to initialize the member variables and to handle the exchange of data between the dialog controls, the member variables, and the properties.

The `m_caption`, `m_circleOffset`, and `m_circleShape` member variables are declared in CIRCPPG.H:

```
class CCircPropPage : public COlePropertyPage
{
    ...
    //{{AFX_DATA(CCircPropPage)
    ...
    CString m_caption;
    int m_circleOffset;
```

```
    BOOL m_circleShape;
    //}}AFX_DATA
    ...
};
```

The member variables are initialized in the constructor for the `CCircPropPage` class, the `CCircPropPage` function in CIRCPPG.CPP:

```
CCircPropPage::CCircPropPage() :
    COlePropertyPage(AfxGetInstanceHandle(), IDD, IDS_CIRCCTRL_PPG_CAPTION)
{
    //{{AFX_DATA_INIT(CCircPropPage)
    m_caption = _T("");
    m_circleOffset = 0;
    m_circleShape = FALSE;
    //}}AFX_DATA_INIT
}
```

Notice that strings assigned to member variables are first passed through the **_T** macro. This is the same macro used for string parameters to **PX_** functions in `CCircCtrl::DoPropExchange`. The **_T** macro is used to maintain compatibility between different string representations, and it must be used for all literal strings in an OLE control project.

Data transfer is handled by the **DDP_** and **DDX_** macros in the `DoDataExchange` function in CIRCPPG.CPP:

```
void CCircPropPage::DoDataExchange(CDataExchange* pDX)
{
    //{{AFX_DATA_MAP(CCircPropPage)
    DDP_Text(pDX, IDC_CAPTION, m_caption, _T("Caption"));
    DDX_Text(pDX, IDC_CAPTION, m_caption);
    DDP_Text(pDX, IDC_CIRCLEOFFSET, m_circleOffset, _T("CircleOffset"));
    DDX_Text(pDX, IDC_CIRCLEOFFSET, m_circleOffset);
    DDP_Check(pDX, IDC_CIRCLESHAPE, m_circleShape, _T("CircleShape"));
    DDX_Check(pDX, IDC_CIRCLESHAPE, m_circleShape);
    //}}AFX_DATA_MAP
    DDP_PostProcessing(pDX);
}
```

The **DDX_** macros are the same macros used for standard MFC dialog boxes. They synchronize dialog controls with dialog member variables. The **DDP_**macros are used only in OLE control property pages. They synchronize property page dialog member variables with specific control properties. Translations between an edit control and a **short** value and between a check box and a **BOOL** value are automatic. Similar to strings assigned to member variables, strings passed as parameters to **DDP_** macros are first passed through the **_T** macro.

The Circle control now has a general property page that can be used to display and modify the values of several of its properties. The `IDC_CAPTION` edit control, `IDC_CIRCLEOFFSET` edit control, and `IDC_CIRCLESHAPE` check box are linked through property page member variables to the Caption, CircleOffset, and

CircleShape properties, respectively. Between the default property page added in this chapter, and the stock color and font property pages added earlier, all of the Circle control properties can be accessed through property pages.

Rebuilding the Control with the Property Page

You should rebuild the Circle control to reflect these latest changes. Because the CIRC.ODL file was not changed, the type library will not be updated as part of the build.

▶ To rebuild the control

- From the View menu, choose Build CIRC.OCX.

Testing the Default Property Page

Use Test Container to verify that the default property page has controls linked to the Caption, CircleOffset, and CircleShape properties.

▶ To test the default property page

1. From the Tools menu, choose OLE Control Test Container.
2. From the Edit menu, select Insert OLE Control.

   The Insert OLE Control dialog box appears.
3. From the Object Type list box, choose Circ Control.
4. Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.
5. From the Edit menu, choose Circ Control, then choose Properties.

   The Circ Control Properties dialog box appears.
6. From the Circ Control Properties drop-down list box, select General. The modified property page appears.
7. Change the Caption, CircleOffset, and CircleShape properties using the property page. Click Apply to see your changes reflected in the control.

CHAPTER 28

Simple Data Binding

Data binding is one of the more powerful features of OLE controls. Data binding is a notification mechanism that links control properties through the container to a data source, such as a database field. In this chapter, a bound property is added to the Circle control to illustrate simple data binding.

Note There is a difference between a bindable property and a bound property. Bindable refers to the fact that a property is available to be bound. A bindable property becomes a bound property at run time, when the control is created and inserted into a container that responds to bound property notifications.

Optimistic and Pessimistic Data Binding

There are two levels of data binding: optimistic and pessimistic. With optimistic data binding, the control assumes that changes can be made to a bound property; with pessimistic data binding, the control is required to ask the container's permission before making changes to the bound property. Whenever a bound property is changed, the control must notify the container by calling the appropriate function, depending on the level of data binding supported.

When optimistic data binding is used, the control notifies the container by calling the **BoundPropertyChanged** function. It is this form of data binding that is used for the Note property in the Circle control.

When pessimistic data binding is used, the control requests permission from the container by calling the **BoundPropertyRequestEdit** function. If the **BoundPropertyRequestEdit** function returns **TRUE**, the control may change the bound property. However, if the **BoundPropertyRequestEdit** function returns **FALSE**, the control must not change the bound property.

Test Container's Notification Log dialog box helps you test bound properties that use either optimistic or pessimistic data binding. When the **BoundPropertyChanged** function is called, a notification is logged in the dialog box. You are also allowed to choose the response to each call to the **BoundPropertyRequestEdit** function. A different container might update a field in a database record when the

BoundPropertyChanged function is called. It might also return **FALSE** from a call to the **BoundPropertyRequestEdit** function if the field or record was in use.

Note You can find the code produced by working through this part of the tutorial in the CIRC3 sample source code directory.

To clearly show the differences between bindable and non-bindable properties, a custom Get/Set property, called Note, will be implemented and changed into a bindable property in two separate steps.

In this chapter you will:

- Define the Note custom Get/Set property.
- Make the Note property bindable.
- Notify the control's container of changes.

Defining the Note Property

Initially, the Note property is a typical custom Get/Set property of type **BSTR**. After the Note property is added to the Circle control, the `GetNote` and `SetNote` functions are completed, the property is made persistent, and a control to edit its value is added to the property page.

▶ To add the Note Get/Set property

1 From the View menu, choose ClassWizard.

2 Choose the OLE Automation tab.

3 In the Class name drop-down list box, select CCircCtrl.

4 Click Add Property.

The Add Property dialog box appears.

5 In the edit control of the External name drop-down combo box, type `Note`.

6 Under Implementation, select Get/Set Methods.

7 From the Type list box, select BSTR.

8 Click OK to close the Add Property dialog box.

This returns you to the OLE Automation tab. Notice that the implementation of the Note property appears as:

```
Implementation:

BSTR GetNote();
void SetNote(LPCTSTR lpszNewValue);
```

9 Click OK to confirm your choices and close ClassWizard.

ClassWizard adds the Note property to the Circle control. As you've seen in earlier chapters, ClassWizard modifies CIRC.ODL, CIRCCTL.CPP, and CIRCCTL.H to define the Note property.

Completing the GetNote and SetNote Functions

To complete the `GetNote` and `SetNote` functions, add the `m_note` member variable to the `CCircCtrl` class to hold the value of the Note property. The `GetNote` and `SetNote` functions will be modified to use this member variable.

Because Note is a persistent property, the `SetNote` function must call the `SetModifiedFlag` function to set the modified flag. A call to the `SetModifiedFlag` function is already included in the `SetNote` function created by ClassWizard. Because the Note property affects the visual appearance of the control, the `SetNote` function must also call the `InvalidateControl` function to redraw the control.

▶ To complete the GetNote and SetNote functions

1 Add the following line to CIRCCTL.H:

```
CString m_note;
```

as shown in the following example:

```
class CCircCtrl : public COleControl
{
...
protected:
    ~CCircCtrl();
    void GetDrawRect(CRect* rc);
    short m_circleOffset;
    BOOL InBounds(short nOffset);
    BOOL InCircle(CPoint& point);
    void FlashColor(CDC* pdc);
   CString m_note;
...
};
```

2 Modify the `GetNote` and `SetNote` functions at the end of CIRCCTL.CPP as shown below:

```
BSTR CCircCtrl::GetNote()
{
    return m_note.AllocSysString();
}

void CCircCtrl::SetNote(LPCTSTR lpszNewValue)
{
    if (m_note != lpszNewValue)
    {
```

```
            m_note = lpszNewValue;
            SetModifiedFlag();
            InvalidateControl();
        }
    }
```

Making the Note Property Persistent

To manage the persistence of the Note property and to initialize it to a default value when the Circle control is created, add a call to a **PX_** function in `DoPropExchange` in CIRCCTL.CPP.

Add the following line to CIRCCTL.CPP:

```
PX_String(pPX, _T("Note"), m_note, _T(""));
```

as shown in the following code example:

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);

    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
    PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
    PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor, RGB(0xFF, 0x00, 0x00));
    PX_String(pPX, _T("Note"), m_note, _T(""));
}
```

Since the Note property is of type **BSTR**, the **PX_String** function is used. The `m_note` member variable (which holds the Note property value) and a zero-length string (which is the default value for the Note property) are parameters to this function. As with all literal strings, the property name string and default value string are passed through the **_T** macro before being passed as parameters to the **PX_String** function.

Displaying the Note Property

To display the Note property, change the drawing behavior in the `OnDraw` function in CIRCCTL.CPP. The Note property is drawn in the upper-left corner of the control's bounding rectangle, using the stock Font and ForeColor properties.

▶ To change the OnDraw function to display the Note property

- Change or add the following lines in the `OnDraw` function in CIRCCTL.CPP:

```
    // Draw the caption and note using the stock Font and ForeColor
properties
...
    pdc->SetTextAlign(TA_LEFT | TA_TOP);
    pdc->ExtTextOut(rc.left, rc.top,
        ETO_CLIPPED, rc, m_note, m_note.GetLength(), NULL);
```

as shown in the following example:

```
void CCircCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    CBrush* pOldBrush;
    CBrush bkBrush(TranslateColor(GetBackColor()));
    CPen* pOldPen;
    CRect rc = rcBounds;
    CFont pOldFont;
    TEXTMETRIC tm;
    const CString& strCaption = InternalGetText();

    // Set the ForeColor property color and transparent background
mode into the device context
    pdc->SetTextColor(TranslateColor(GetForeColor()));
    pdc->SetBkMode(TRANSPARENT);

    // Paint the background using the BackColor property
    pdc->FillRect(rcBounds, &bkBrush);

    // Draw the ellipse using the BackColor property and a black pen
    GetDrawRect(&rc);
    pOldBrush = pdc->SelectObject(&bkBrush);
    pOldPen = (CPen*)pdc->SelectStockObject(BLACK_PEN);
    pdc->Ellipse(rc);

   // Draw the caption and note using the stock Font and ForeColor
properties
    pOldFont = SelectStockFont(pdc);
    pdc->GetTextMetrics(&tm);
    pdc->SetTextAlign(TA_CENTER | TA_TOP);
    pdc->ExtTextOut((rc.left + rc.right) / 2, (rc.top + rc.bottom -
tm.tmHeight) / 2,
        ETO_CLIPPED, rc, strCaption, strCaption.GetLength(), NULL);
   pdc->SetTextAlign(TA_LEFT | TA_TOP);
   pdc->ExtTextOut(rc.left, rc.top,
       ETO_CLIPPED, rc, m_note, m_note.GetLength(), NULL);
    pdc->SelectObject(pOldFont);

    pdc->SelectObject(pOldPen);
    pdc->SelectObject(pOldBrush);
}
```

Adding the Note Property to the Default Property Page

Add the Note property to the default property page to allow Circle control users to change its value.

First, use the resource editor to add an edit control to the default property page.

▶ To add an edit control to the default property page

1 In the ResourceView pane, open the Circ project folder.

2 Open the Dialog folder.

3 Open IDD_PROPPAGE_CIRC to edit the property page template.

4 Select the Static Text tool in the Control Palette and place a static text control in the dialog.

5 Double-click the static text control to bring up the Test Properties dialog box.

6 Using the Test Properties dialog box, change the static text control's caption to &Note:

7 In the Control Palette, select the Edit Box tool in the Control Palette and place an edit box control next to the static text control in the dialog.

8 Double-click the edit box control to bring up the Test Properties Dialog box.

9 Using the Test Properties dialog box, change the edit control's ID to IDC_NOTE.

10 Save the changes to the dialog and exit the resource editor.

Link the Note property to the new edit control in the default property page using a shortcut to the Add Member Variable dialog box in ClassWizard.

▶ To link the edit control with the Note property

1 In the ResourceView pane, open the Circ project folder.

2 Open the Dialog folder.

3 Open the IDD_PROPPAGE_CIRC entry in the Dialog folder to load the property page template.

4 While holding down the CTRL key, double-click the edit box control for the Note property.

This automatically brings up the ClassWizard Add Member Variable dialog box.

5 Type `note` into the Member variable name edit control, after the m_ that is already there, so that the edit control contains m_note.

6 In the Category drop-down list box, select Value.

7 In the Variable type drop down list box, select CString.

8 Type `Note` into the Optional OLE property name drop down combo box.

9 Click OK to confirm your choices and close the Add Member Variable dialog box.

If you were to open ClassWizard , the Member Variable tab would contain the new member variable mapping for the Caption property.

```
Control IDs:        Type       Member

IDC_CAPTION         CString    m_caption
IDC_CIRCLEOFFSET    int        m_circleOffset
IDC_CIRCLESHAPE     BOOL       m_circleShape
IDC_NOTE            CString    m_note
```

ClassWizard adds the new `m_note` member variable to the `CCircPropPage` class. ClassWizard also modifies the `DoDataExchange` function in the `CCircPropPage` class.

Making the Note Property Bindable

At this point, the Note property is a fully implemented, normal Get/Set property. The next step is to use the OLE Automation tab in ClassWizard to make the Note property a bindable property.

▶ To make the Note property bindable

1. From the View menu, choose ClassWizard.
2. Choose the OLE Automation tab.
3. In the Class name drop down list box, select CCircCtrl.

   In the External name list box, select Note.
4. Click Data Binding.

   The Data Binding dialog box appears.
6. Check the Bindable Property check box.

   This specifies that this is a bindable property and that the container will be notified of all changes to it.
7. Click OK to confirm the data binding settings and close the Data Binding dialog box.
8. Click OK to confirm your choices and close ClassWizard.

ClassWizard changes the Note property definition in the file CIRC.ODL so that the type library includes the information that the Note property is bindable.

Notifying the Container of Changes

A control must notify the container when changes are made to a bound property by calling the `BoundPropertyChanged` function.

Because Note is a Get/Set property, all changes to the property are confined to the `SetNote` function, which is where `BoundPropertyChanged` is called.

▶ To modify the SetNote function to notify the container about changes to the Note property

- Add the following line to the SetNote function in CIRCCTL.CPP:

```
BoundPropertyChanged(dispidNote);
```

as shown in the following example:

```
void CCircCtrl::SetNote(LPCTSTR lpszNewValue)
{
    if (m_note != lpszNewValue)
    {
        m_note = lpszNewValue;
        SetModifiedFlag();
        InvalidateControl();
       BoundPropertyChanged(dispidNote);
    }
}
```

The call to `BoundPropertyChanged` is added after the `m_note` member variable is updated, and all other actions involved in changing the Note property are completed.

The single parameter to `BoundPropertyChanged`, `dispidNote`, is the dispatch ID for the Note property. This parameter is defined in an enumeration in CIRCCTL.H:

```
// Dispatch and event IDs
public:
    enum {
    //{{AFX_DISP_ID(CCircCtrl)
    ...
    dispidNote = 4L,
    ...
    //}}AFX_DISP_ID
    };
```

Rebuilding the Control with Data Binding Support

Now that a property with data binding support has been added, you need to rebuild the control.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the Control Data Binding Changes

Because the Note property uses optimistic data binding, the Circle control assumes that all changes to the Note property are allowed. The `SetNote` function simply changes the Note property and notifies the container by calling the `BoundPropertyChanged` function. This notification is displayed in Test Container's Notification Log dialog box.

▶ To test the control data binding changes

1 From the Tools menu, choose OLE Control Test Container.

2 From the View menu, choose Notification Log.

The Notification Log dialog box appears.

The option buttons at the bottom of the Notification Log dialog box can be used to test control properties that use pessimistic data binding. However, the Note property uses optimistic data binding, so they are not used here.

3 From the Edit menu, choose Insert OLE Control.

The Insert OLE Control dialog box appears.

4 From the Object Type list box, select Circ Control.

5 Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.

Notice that the Notification Log dialog box is empty when the control is first created.

6 From the Edit Menu, choose Embedded Object Functions and then Properties.

The Circ Control Properties dialog box appears.

7 Enter a new value for the Note property, and click Apply.

The Note property is displayed in the upper-left corner of the control.

Information about the change to the Note property appears in the Notification Log dialog box as shown below:

```
00_Circ_Control: 'Note' changed
```

This kind of information appears in the Notification Log dialog box every time the `BoundPropertyChanged` function is called for a bound property. The line shows the control number and type, and the property name, and states that the property has changed.

8 Change the Note property several more times and watch the Notification Log dialog. Close Test Container when you are finished.

Using the `BoundPropertyChanged` function to notify the container of changes to bound properties illustrates the simplest level of data binding. For more information on the other options in the Data Binding dialog box, see the article "OLE Controls: Using Data Binding in an OLE Control" in *Programming with MFC*.

CHAPTER 29

Versions and Serialization

Because different versions of an OLE control might have different properties, support is included in the OLE control classes for handling different versions of persistent data. In this chapter, the Circle control is modified to reject persistent data that is of a different version than the control itself. Selective loading and/or storing of persistent data for different versions of controls is also discussed.

Note You can find the code produced by working through this part of the tutorial in the CIRC3 sample source code directory.

In this chapter, you will:

- Learn about serialization of control version information.
- Modify the Circle control so that it ignores other versions of persistent data.
- Learn how a control can serialize earlier versions of persistent data.

Serialization of Control Version Information

Serialization is the process by which persistent data is loaded or stored through an exchange object. One type of exchange object is used to initialize a control's persistent properties to default values; another type is used to load and store persistent data in persistent storage. A control's persistent data is serialized in the **DoPropExchange** function.

The Circle control's version, stock properties, and persistent custom properties (in that order) are serialized in the `DoPropExchange` function in CIRCCTL.CPP.

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
```

```
    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
    PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
    PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor, RGB(0xFF, 0x00, 0x00));
    PX_String(pPX, _T("Note"), m_note, _T(""));
}
```

When the Circle control project was created, ControlWizard produced the code that calls the `ExchangeVersion` and **COleControl::DoPropExchange** functions. ControlWizard also defined the global constants `_wVerMajor` and `_wVerMinor` in CIRC.CPP as 1 and 0 respectively, which represents version 1.0.

The `ExchangeVersion` function serializes a control's version and sets the version used by the exchange object, `pPX`. This call should always be made before any version-sensitive persistent data is serialized. When persistent data is being initialized or written into persistent storage through the exchange object, the exchange object's version is set to the version parameter passed to the function. When persistent data is being read from persistent storage through the exchange object, the exchange object's version is read from persistent storage.

The **COleControl::DoPropExchange** function serializes all of a control's stock properties. The **PX_** functions serialize each of the Circle control's persistent custom properties. These calls were added in previous chapters of the tutorial.

Serializing Different Versions of Persistent Data

The control writer knows which properties earlier versions of the control have and in what order they are serialized. Rather than ignoring the exchange object, the persistent custom properties of the old version of the control can be loaded or stored through the exchange object in a selective manner.

When an earlier version of persistent data is to be loaded, any new properties can be initialized to their default values.

Note It is not recommended that new versions of a control remove features (such as properties) of earlier versions.

When an earlier version of persistent data is to be stored, any new properties can be skipped.

Consider a hypothetical version 2.0 of the Circle control that has a new custom property called BorderWidth. The BorderWidth property is of type **short** and has a default value of 1.

It is assumed that all versions of the Circle control will have all the properties of earlier versions, and that the properties are stored in the same order as in earlier versions. This is a recommendation for all OLE controls.

Here is what the version 2.0 DoPropExchange function might look like:

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor), FALSE);
    COleControl::DoPropExchange(pPX);
    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
    PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
    PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor, RGB(0xFF,
        0x00, 0x00));
    PX_String(pPX, _T("Note"), m_note, _T(""));

    if (pPx->GetVersion() >= (DWORD)MAKELONG(_wVerMinor, _wVerMajor))
        PX_Short(pPX, _T("BorderWidth"), m_borderWidth, 1);
    else
        if (pPx->IsLoading())
          m_borderWidth = 1;
}
```

Notice the third parameter to the ExchangeVersion function. This is an optional parameter that specifies whether or not the control should store the same version of persistent data as was last loaded. By passing **FALSE**, the old version is maintained. The default for this parameter is **TRUE**, which specifies that the control should always store the current version of persistent data, no matter what version of persistent data is loaded.

All stock properties and custom properties from versions before 2.0 are serialized as usual, in the same order as they were in the earlier versions.

If the exchange object's version is greater than or equal to 2.0, the BorderWidth property is serialized as usual.

If the exchange object's version is less than 2.0, and the IsLoading function returns **TRUE**, the BorderWidth property is initialized to its default value.

Ignoring Different Versions of Persistent Data

When a control is requested to load persistent data through an exchange object with a version different from the control, it must do something reasonable to set the values of its persistent custom properties. The simplest approach is to ignore the exchange object and initialize each persistent custom property to its default value by modifying the Circle control's DoPropExchange function. The changes involve using two member functions of the exchange object, GetVersion and IsLoading.

The GetVersion member function returns the exchange object's version. The exchange object's IsLoading member function returns **TRUE** if the control is to load values through the exchange object; it returns **FALSE** if the control is to store values through the exchange object.

The `GetVersion` function is used to compare the exchange object's version and the control's version. If the versions are the same, each persistent custom property is serialized as usual. If the versions are different and the `IsLoading` function returns **TRUE**, the Circle control's persistent properties are initialized to their default values. If the versions are different and the `IsLoading` function returns **FALSE**, the Circle control ignores the exchange object. Technically, this situation never occurs, because the Circle control always stores the current version of persistent data.

Modify the `DoPropExchange` function to look like this:

```
void CCircCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    if (pPX->GetVersion() == (DWORD)MAKELONG(_wVerMinor, _wVerMajor))
    {
        PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
        PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
        PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor, RGB(0xFF,
        0x00, 0x00));
        PX_String(pPX, _T("Note"), m_note, _T(""));
    }
    else
        if (pPX->IsLoading())
        {
            m_circleShape = TRUE;
            m_circleOffset = 0;
            m_flashColor = RGB(0xFF, 0x00, 0x00);
            m_note = _T("");
        }
}
```

As described earlier, the `ExchangeVersion` function sets the exchange object's version. This is the version of persistent data that the control is to load or store through the exchange object. This version is returned by the `GetVersion` function.

The control version and stock properties are serialized as usual. The **COleControl::DoPropExchange** function correctly loads or stores stock property values for different versions because part of what is serialized is information about which stock properties were stored for that version.

If the exchange object's version and the control's version are the same, the Circle control's persistent custom properties are loaded or stored through the exchange object as usual.

If the versions are different, and persistent data is to be loaded, all persistent properties are initialized to their default values instead.

If the versions are different, and persistent data is to be stored, the exchange object is ignored and no persistent data is stored through the exchange object. This case should never happen with the Circle control because the exchange object's version and the control's version may be different only when data is to be loaded through the

exchange object. Another form of the `ExchangeVersion` function, which allows a control to specify that persistent data should be stored as the same version that was last loaded, is shown in "Testing the Control."

Rebuilding the Control with Version Support Implemented

Now that version support has been added, you need to rebuild the control.

▶ To rebuild the control

- From the Build menu, choose Build CIRC.OCX.

Testing the Control

Testing different versions of the same control is not easy, simply because only one of the versions can be registered at a time. This is one way in which you might check to see if a newer version of the Circle control correctly ignores persistent data stored by an earlier version of the control.

▶ To test the version checking serialization code

1. From the Tools menu, choose OLE Control Test Container.
2. From the Edit menu, choose Insert OLE Control.

   The Insert OLE Control dialog box appears.
3. From the Object Type list box, select Circ Control.
4. Click OK to close the Insert OLE Control dialog box and insert the control into Test Container.
5. From the Edit menu, choose Edit, then Embedded Object Functions, then Properties, to show the Circle control's property page.
6. Change the value of the CircleShape property to **FALSE** (clear the check box and then click OK).
7. From the File menu, choose Save to Stream or Save to Substorage.
8. From the Edit menu, choose Delete All.
9. From Microsoft Developer Studio, in the file CIRC.CPP, change the version number to 2.0 by changing the value of the `_wVerMajor` global constant to 2.
10. Choose Build CIRC.OCX to rebuild the control.
11. From Test Container's File menu, choose Load.
12. From the Edit menu, choose Edit then Circ Control Object, then Properties to show the Circle control's property page.

13 View the value of the CircleShape property in the property page.

The persistent data saved in step 7 is version 1.0. When the control is recreated in step 11, the control's version is 2.0. Because the persistent data and the control have different versions, the control's persistent properties are initialized to their default values. The stored values are ignored. In particular, the CircleShape property should be **TRUE** (the check box checked).

14 Change the value of the CircleShape property to **FALSE** again (clear the checkbox, then click OK).

15 From the File menu, choose Save to Stream or Save to Substorage.

16 From the Edit menu, choose Delete All.

17 From Test Container's File menu, choose Load.

18 From the Edit menu, choose Edit, then Circ Control Object, then Properties, to show the Circle control's property page.

19 View the value of the CircleShape property using the property page.

The persistent data saved in step 15 is version 2.0. When the control is recreated in step 17, the control's version is 2.0. Because the persistent data and the control have the same version, the control's persistent properties are set to the stored values. In particular, the CircleShape property should be **FALSE** (the check box cleared).

If you want your Circle control to match the control found in the sample directories, be sure to restore the control version to 1.0 (change the value of `_wVerMajor` in CIRC.CPP to 1 and rebuild the control.)

PART 4

The Database Tutorials

CHAPTER 30

Creating a Database Application

This tutorial shows you how to develop a form-based Microsoft Foundation Class Library (MFC) database application. You'll learn how to:

- Use AppWizard to create the skeletal database application.
- Create and use **CRecordset** objects to open tables and run queries.
- Create and use **CRecordView** objects for form-based applications.
- Use database support within the framework's document/view architecture.
- Add, update, and delete records.
- Manage multiple tables.
- Handle database exceptions.

Important This tutorial assumes you are familiar with Visual C++ and the Microsoft Foundation Class Library. If you aren't, try the Scribble tutorial in Chapters 2 through 11 before you begin this tutorial. The Scribble tutorial introduces important class library concepts and techniques and teaches you to use the wizards and the resource editors.

The Tutorial Example: Enroll

The tutorial example program, Enroll, manages a student registration database similar to, but simpler than, a college registration system. It will help you to follow the tutorial if you understand the structure of the student registration database.

Enroll is based on a database, STDREG32.MDB, that you will register with ODBC as the "Student Registration" data source name. Table 30.1 lists the tables, what they store, and the columns in them.

Table 30.1 Tables in the Student Registration Database

Table name	Contents	Column list
Course	Think of each record as an entry in a course catalog. Example: the MATH101 course.	CourseID* CourseTitle Hours
Section†	A section record is a specific offering of a course at a specific time. For example, MATH101 may have many sections.	SectionNo* CourseID* InstructorID Schedule RoomNo
Student	A record for each student at the school.	StudentID* Name GradYear
Enrollment	A record for each student in a particular section of a course. For a given student, there is an enrollment record for each course the student is taking.	CourseID* SectionNo* StudentID* Grade
Instructor	A record for each instructor at the school.	InstructorID* Name RoomNo

*Indicates the column (or columns) that comprise the table's primary key.

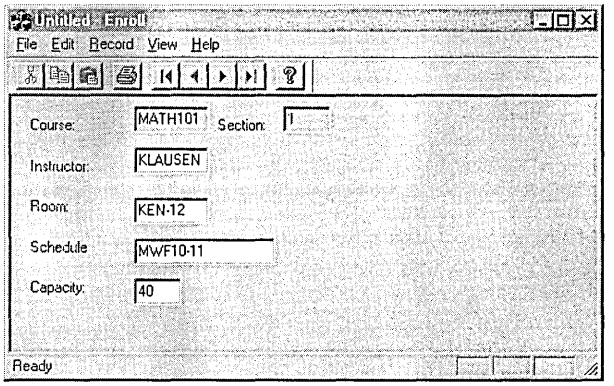
†The Dynabind_Section table is used in the Dynabind sample, but not in the Enroll tutorial.

You can use Books Online to copy STDREG32.MDB to your local drive. You can examine it, add records, and so on, using Microsoft Access.

Note You can also easily install the sample source project files for Enroll. For more information, see "Installing the Sample Files."

Enroll lets you use a "form" — a view with dialog-style controls — to view registration information for courses, section by section. Section information displayed includes the course name, section number, instructor, room, and schedule (such as "MWF 10-11"). For example, you can view section 1 of the course MATH 101, then section 2, and so on. The initial tutorial step provides read-only viewing of all sections. Later steps add more capabilities, including updates. Figure 30.1 shows what the Enroll application looks like at the end of the tutorial.

Figure 30.1 The Enroll Tutorial Application



Setting Up the Student Registration Data Source

Before you start the Enroll tutorial, you need to set up the Student Registration database and register it as an ODBC data source. Choose a database format for which you have the corresponding database management system (DBMS) and 32-bit ODBC driver. Microsoft Developer Studio ships 32-bit ODBC drivers for most standard database formats, including: SQL Server, Access, Paradox, dBase, FoxPro, Excel, Oracle and Microsoft Text.

- If you want to use MFC database support for SQL Server, you need the SQL Server product in addition to the ODBC driver for SQL Server that is provided with Developer Studio.
- If you want to use other database formats, you need the DBMS as well as the ODBC driver.
- If you want to use the Microsoft Access database format, you need only the 32-bit Microsoft Access ODBC driver to create a database schema. This driver is installed automatically when you run a Typical setup. This is an exception to the requirements listed above. You may find it helpful, however, to use Microsoft Access itself in conjunction with MFC database support, as it will facilitate working interactively with your database schema and data.

To set up the student registration database you must:

- Specify a database.
- Register the database with ODBC.

If you are using a DBMS other than the prebuilt Microsoft Access STDREG32.MDB database file, you need to add tables to your database so that it matches the Student

Registration database schema. Additionally, you may need to install drivers other than those that Visual C++ Setup installs for you. In this case, perform the following procedure(s):

- Use the STDREG.EXE tool to add tables to the Student Registration database.
- Install additional 32-bit ODBC drivers for your DBMS.

  This step is not necessary if your DBMS uses the dBase, FoxPro, Access or SQL Server drivers, as Setup installs them automatically when you choose the Typical installation option. Perform this step to install additional ODBC drivers, including those supplied by Developer Studio: Paradox, Microsoft Text, Excel or Oracle.

Each of these procedures is described in the following topics.

Specify a Database

The easiest way to supply a database for the Enroll tutorial is to use the pre-built STDREG32.MDB Microsoft Access database file, included for this purpose. Alternatively, you can create your own database.

▶ To use STDREG32.MDB

1. From InfoView, expand the following folders:

   Samples \ MFC Samples \ Database samples (ODBC and DAO)
2. Double-click the page node for the STDREG sample.
3. In the STDREG topic, click the button provided to copy sample project files.
4. In the Sample Application dialog box, select STDREG32.MDB and choose Copy.
5. In the Copy dialog box, navigate to the directory where you want to copy this file, and click OK.

   Developer Studio creates this directory for you, if necessary, and copies the file.

STDREG32.MDB already contains the tables and records used in the tutorial. If you use this file, you do not need to use the STDREG.EXE tool to add any tables.

▶ To create your own database

- Create a new database schema using the database administration capability of your DBMS.

  Depending on the type of DBMS, you might create the new database on a server that is different from the PC where you will be doing MFC database development. In either case, you need to use the STDREG.EXE tool to add tables to the new database.

Register the New Database with ODBC

You'll register the new database with the ODBC data source name "Student Registration." This data source name (DSN) is referred to by the Enroll application. You must register the database even if you are using the pre-built STDREG32.MDB Microsoft Access database file.

You can register the data source by using ODBC Administrator from the Control Panel, or by running STDREG.EXE.

Note The STDREG tool is provided for use with this tutorial as a convenient method for registering data sources with ODBC and for populating databases with the appropriate tables and data. Normally, you will use ODBC Administrator from the Control Panel to register your data sources with the appropriate tables and data if you are not going to use the pre-built STDREG32.MDB database.

The following two procedures describe how to register the data source if you are using the Microsoft Access ODBC driver. If you are using another driver, the basic procedure will vary somewhat. For more information, refer to the ODBC SDK topic, "Adding Data Sources."

▶ To register the data source by using ODBC

1. Open Control Panel, and double-click the ODBC icon.
2. In the Data Sources dialog box, choose Add.

   The Add Data Source dialog box appears.
3. Choose the driver you want to use with your database, in this case, Microsoft Access, and click OK.

   The ODBC Setup dialog specific to the driver you specified appears.
4. In the Data Source Name box, type `Student Registration`.
5. Optionally, enter a description for the database.
6. In the Database group box, click the select button and then navigate to the location of STDREG32.MDB
7. Click OK to select the database.
8. Click OK to exit the ODBC Setup dialog, and click Close to exit the Data Sources dialog.

For more information about using the ODBC Administrator, see the encyclopedia articles" ODBC Administrator" and "Data Source (ODBC)," in *Programming with MFC*.

▶ To register the data source by using STDREG.EXE

1 First run STDREG.EXE, by using the following procedure:

- From InfoView, expand the following folders:

  Samples \ MFC Samples \ Database Samples (ODBC and DAO)
- Double-click the page node for the STDREG sample.
- In the STDREG topic, click the button provided to copy sample project files.
- In the Sample Application dialog box, select STDREG.EXE and choose Run.

2 Choose Add Data Source.

3 Follow the instructions in steps 3 through 7 of the previous procedure.

Use STDREG.EXE to Add Tables

If you are using a DBMS other than the prebuilt Microsoft Access STDREG32.MDB database file, use the STDREG tool to add tables to the Student Registration database. This tool creates the Student Registration tables listed in Table 30.1. The tool also adds records to the newly created tables for use as test data by the Enroll application.

The STDREG.EXE sources illustrate how to directly send SQL statements such as **CREATE TABLE**, and how to use ODBC catalog functions such as **SQLGetTypeInfo**.

Note This procedure assumes you have used STDREG.EXE to register the Student Registration data source.

▶ To add tables by using STDREG.EXE

1 If necessary, start STDREG.EXE, and then choose the Initialize Data option.

Depending on the type of database you are using, you may need to respond to a login dialog box, such as the SQL Server Login dialog box.

2 After logging in to the Student Registration data source, respond to a series of three Enter SQL Column Syntax dialog boxes.

After you respond to three successive Enter SQL Column Syntax dialog boxes, STDREG creates the tables in the new database.

3 When STDREG has completed this task, choose Exit.

You can rerun the STDREG tool at any time to remove and recreate the tables in the Student Registration data source.

For more information about using STDREG.EXE, see the "STDREG Sample" topic.

Installing Additional 32-Bit ODBC Drivers for Your DBMS

You need only install the ODBC driver for your DBMS once, and you can use it with more than one data source. If you choose the Typical Setup option, Setup provides MFC Database Support and installs the ODBC drivers for Access, dBase, FoxPro and SQL Server automatically. You can install the additional drivers provided with Visual

C++ (Text, Paradox, Excel and Oracle) by running Setup again, as described in the following procedure.

If you want to install other drivers not shipped with Visual C++, refer to the documentation that came with your driver. You can also review the ODBC SDK documentation, in particular, the "Installing Drivers" topic.

▶ To install additional drivers provided with Visual C++

1. Run Setup, and under Installation Options, select Custom.
2. Click Next, and in the Microsoft Visual C++ Setup dialog, uncheck each option except Database Options.
3. Highlight Database Options and then click the Details button.
4. In the Database Options dialog, highlight Microsoft ODBC Drivers, and again Click the Details button. If you do not want to install DAO database support, clear the Microsoft Data Access Objects check box.
5. In the Microsoft ODBC Drivers dialog, check the boxes next to any additional drivers you want to install.
6. Click OK, and then click Next to start the installation.

Tutorial Steps

The tutorial consists of three steps. The following table describes the steps briefly.

Tutorial Step	Chapter	Description
1	31	Use AppWizard to create an application with database support. The document embeds a **CRecordset** object for the Section table of the Student Registration data source. Use the dialog editor to design the form. Use ClassWizard to bind controls on the form to fields in the recordset.
2	32	Provide a combo box control on the form so the user can select a course and view its sections. Fill the combo box from a recordset object representing the Course table. Filter and parameterize the recordset to constrain the records it selects.
3	33	Implement a user interface for adding, updating, and deleting records. Handle database exceptions.

The ENROLL sample program directory contains a subdirectory for each step of the tutorial, named STEP1, STEP2, and STEP3. Each step's subdirectory contains complete source files, and other files needed for the step.

If you do not have local copies of the Sample files, you can easily install them. For more information, see "Installing the Sample Files." While you don't need to have a copy of these files locally in order to develop the project yourself, you might find it useful, for example to compare source code.

Enroll Step 4: A Preview

The sample source code for ENROLL includes a fourth step, not covered in the tutorial. Step 4 illustrates additional class library database programming techniques, summarized below. See ENROLL in SAMPLES \ MFC Samples \Tutorials under Samples\ MFC Samples \Tutorials in Books Online for a discussion of Enroll Step 4. The main techniques illustrated by Enroll Step 4 are:

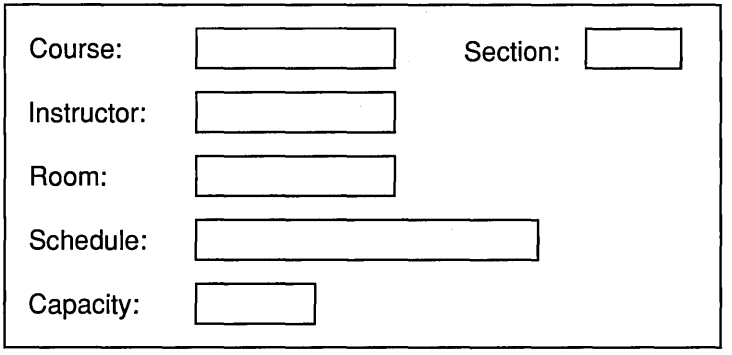
- Using multiple record view classes.
- Switching views in a frame window.
- Using the document object to coordinate multiple forms via **UpdateAllViews** and update hints.

CHAPTER 31

A Simple Form

This tutorial step implements an updatable database form that lets the user examine the records in the Section table one record at a time. You'll create a form that looks like the one shown in Figure 31.1.

Figure 31.1 Enroll's Section Form



This chapter explains:

- Creating the Enroll application.
- Examining the Enroll Step 1 classes.
- Customizing Enroll's database form.
- Binding Enroll's form controls to recordset fields.
- Building and running Enroll Step 1.

If you choose to work along with the tutorial, perform all the steps in the procedures in this chapter. At the end, you'll be able to build and run the Enroll Step 1 application.

About Step 1

Step 1 teaches the basics of:

- Using AppWizard to create an application with database support.
- Using ClassWizard, ClassView and the resource editors to bind controls on a form to data.
- Using recordsets.
- Using record views.

A recordset object represents a set of records selected from a data source. The recordset may represent a selection of one or more specified columns from rows of one or more database tables. A **CRecordset** object represents both (a) this selection of records and (b) the actual field values for one currently selected record. For more information, see "Recordset (ODBC)" in *Programming with MFC.*

A record view is a specialized view class that uses controls laid out in a dialog template resource to view and/or edit the fields of a recordset in a dialog-like form. A **CRecordView** object is associated with both (a) a recordset object and (b) a dialog template resource. The dialog template resource has an ID of the form **IDD_XXX_FORM**, where **XXX** is based on the project name. **CRecordView** derives its form behavior from class **CFormView**. **CRecordView** supports end-user navigation through records, one at a time, using Move First, Move Next, Move Previous, and Move Last commands of the associated **CRecordset** object. When you update the value in a control on the form and navigate to another record, the corresponding recordset field is automatically updated.

While in AppWizard, you identify an Open Database Connectivity (ODBC) data source and a table in the data source. AppWizard creates a pair of classes: a recordset class and a record view class.

For more information, see the articles "Recordset (ODBC)," "Record Views," "AppWizard: Database Support," "Data Source (ODBC)," and "ODBC" in *Programming with MFC.*

Creating a New Database Application

For more information on how to use the Database Options page in AppWizard when you're creating your starter application, see "AppWizard: Database Support" in *Programming with MFC.* AppWizard lets you specify whether your database application uses a file as well as a database. The Enroll application doesn't need a file, so it is based on the "Database view without file support" option in AppWizard.

For more information about applications that don't use file support, see "Serialization: Serialization vs. Database Input/Output" in *Programming with MFC*

Note In order to successfully complete the following procedure, you must have completed the steps required to register the Sudent Registration database with ODBC. If you have not done so, see "Setting Up the Student Registration Data Source," in Chapter 30 of this book.

▶ To create the tutorial database application

1 From the File menu, choose New.

The New dialog box appears.

2 Select Project Workspace.

The New Project Workspace dialog box appears.

3 In the Name box, type `Enroll`.

AppWizard creates a project directory with this name under the main (root) directory specified in the Location box.

4 In the Type list box, make sure MFC AppWizard (exe) is specified.

5 If necessary, use the Location box to specify a different root directory for the Enroll project files that AppWizard creates under the Enroll project directory.

6 If any check boxes other than Win32 appear in the Platforms box, clear them.

7 Click Create.

AppWizard creates the project directory, and the MFC AppWizard–Step 1 dialog box appears.

8 Choose the Single Document radio button, and then click Next to continue to the Step 2 (database options) dialog box.

When you create a database application without file support, AppWizard always creates it as an SDI application.

9 In the AppWizard–Step 2 dialog box:

- Select the "Database view without file support" option.

  This enables the Data Source button.

- Click Data Source.

  The Database Options dialog box appears.

- Select ODBC and, from the drop-down list box, select Student Registration.

  Depending on the database type, you may need to supply additional information to log into the data source.

- Choose OK (there's no need to change the other default options in this dialog).

  The Select Database Tables dialog box appears.

- Select the table name SECTION, and choose OK.

  This returns you to the AppWizard–Step 2 dialog box.

  Depending on the data source type you are using, additional qualifiers may precede or follow the table name.

10 Click Next to proceed to the rest of the AppWizard dialog boxes, and click Next in the AppWizard dialog boxes for Steps 3, 4, and 5 to accept the default options.

   In the AppWizard–Step 6 dialog box, you can check and, if necessary, modify the default names that AppWizard creates for your program's classes and files.

   Note By default, AppWizard bases the names of classes on the project name you supply. This naming is probably fine if your application has only one recordset/view pair. If your application has multiple recordsets and record views, it's a good idea to change the name of the first recordset/view pair created by AppWizard so the naming better reflects the name of the table in the data source. For Enroll, you'll modify two class names and their related header and implementation file names, even though the tutorial uses only one recordset/view pair.

11 In the Step 6 dialog box, make the following changes to class names:

- Select the class `CEnrollSet`, and change its name to `CSectionSet`. Change the header filename to SectionSet.h. Change the implementation file name to SectionSet.cpp.

  The base class is **CRecordSet**. The edit item is disabled to show that you can't change it.

- Select the class `CEnrollView`, and change its name to `CSectionForm`. Change the header filename to SectionForm.h. Change the implementation file name to SectionForm.cpp.

  The base class is **CRecordView**.

12 Choose Finish.

   The New Project Information dialog box appears, summarizing the settings and features AppWizard will generate for you when it creates your project.

   You might want to take a moment to examine the application type, classes, and features that AppWizard automatically provides.

13 Click OK in the New Project Information dialog box.

   AppWizard creates all necessary files, and opens the project.

You can view the classes that AppWizard just created in ClassView. The next section, "Examining the Step 1 Classes," describes this in more detail.

Examining the Step 1 Classes

Once AppWizard creates the Enroll project, you can use ClassView to see a graphical representation of the classes and any default member functions AppWizard created.

You can also use ClassWizard to view member variable bindings that AppWizard specifies for you.

The CSectionSet Recordset Class

The following procedure describes how to view the new recordset class, `CSectionSet`. After examining `CSectionSet`, you'll use the text editor to examine the source files for classes `CSectionForm` and `CEnrollDoc`.

▶ To examine the new recordset class

1. In ClassView, expand the Enroll folder.

   Notice the rich set of classes AppWizard created for you automatically to support the Enroll application.

2. From the list of classes, choose CSectionSet.

   ClassView displays all the member variables that AppWizard created for you, including a variable for each of the Section table's columns. You can use ClassWizard to view how AppWizard has bound the Section table's columns to these member variables.

3. From the View menu, choose ClassWizard.

4. Choose the Member Variables tab.

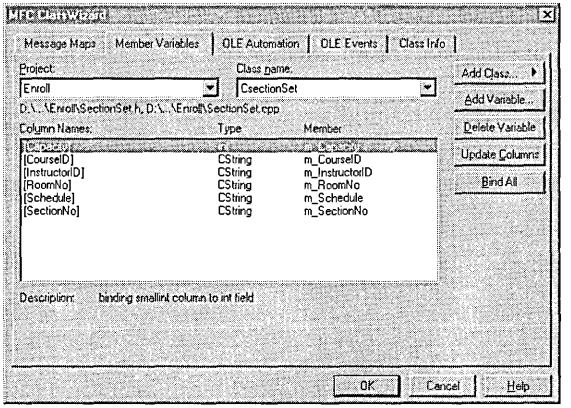
5. In the Class Name box, select CSectionSet.

   What you see in this tab corresponds to the variables displayed in ClassView. AppWizard has bound all of the table's columns to member variables of the `CSectionSet` class. These member variables are called "field data members." AppWizard names the data members automatically, based on the column names from the data source. AppWizard also assigns the correct C++ or class library data type to the data members, based on the column type. In this example, all of the columns are text columns, mapped to type **CString**, except the Capacity column, which is an **int**.

6. Once you've finished examining the `CSectionSet` recordset class, click OK to exit ClassWizard.

Figure 31.2 shows what you see in ClassWizard's Column Names box.

Figure 31.2 Table Columns Mapped to Recordset Data Members



For this tutorial, you will need all of the column bindings. However, in your own application, if you don't want all of a table's columns bound to your recordset, you can delete the recordset field data members for those columns you don't want; from within ClassWizard, select the data member and click the Delete Variable button.

Caution Don't delete any fields that are part of the table's primary key (in this case, the SectionNo and CourseID fields).

In your own application, if you want to change the name of a field data member, use ClassWizard to delete the member and add it again with the new name. For more information, see "ClassWizard: Binding Recordset Fields to Table Columns" in *Programming with MFC*.

In the next procedure, you'll examine the `CSectionForm` record view class from inside the text editor.

The CSectionForm Record View Class

▶ To examine the source code for class CSectionForm,

- Use ClassView to jump to the `OnInitialUpdate` member function of class `CSectionForm`.

  SectionForm.cpp opens inside the text editor, with your cursor just before the `OnInitialUpdate` member function:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;
    CRecordView::OnInitialUpdate();
}
```

The base class framework function **CRecordView::OnInitialUpdate** opens the database if not already open, then opens the recordset, and initializes the form by calling **CFormView::OnInitialUpdate**.

For now, the "form" represented by class `CSectionForm` is empty of controls. Later, in "Customizing the Dialog Template for the Section Form," you'll use the dialog editor to design the form and to map controls on the form to the recordset.

The CEnrollDoc Document Class

In ClassView, you can see that AppWizard created a class, `CEnrollDoc`, derived from **CDocument**.

▶ To view the document class

- In ClassView, double-click the icon for class `CEnrollDoc`.

  This opens the file EnrollDoc.h in the text editor, placing your cursor beside the `CEnrollDoc` class declaration.

What is the role of a document in a database application? In most other applications, the document stores data and serializes it to a file on disk. Often the application reads the whole file into memory at once and writes it back to disk as a whole. In a database application, however, the data is stored in the database, and the end user usually views the data as records. Such an application doesn't need a file.

A document in a database application, then, isn't normally used for its serialization support. So why does Enroll have a document class?

The following code, at the beginning of EnrollDoc.h, reveals that the role of the document class in Enroll is to own the recordset.

```
class CEnrollDoc : public CDocument
{
    ...
    // Attributes
    public:
        CSectionSet m_sectionSet;
    ...
};
```

The recordset object, `m_sectionSet`, is embedded in the document object. Therefore, the recordset object is automatically constructed when the document object is constructed, and automatically deleted when the document object is deleted.

The document class can own any number of recordset objects in this way. For example, Step 4 of Enroll adds a second form and corresponding recordset; the document embeds this second recordset.

In a sense, then, the document class is a proxy for the database. This approach isn't strictly necessary, but if you (or AppWizard) design your database application to use the document class this way, you can better take advantage of the framework's

document/view architecture. For example, if you have multiple views (forms) simultaneously showing some of the contents of the database, you can take advantage of the **CDocument::UpdateAllViews** mechanism to conveniently notify all views about an update that might have been initiated in one of the views.

If you look at the menu resource that AppWizard created when you chose the option "Database view without file support," you'll see that there are no New, Open, Save, or Save As commands on the File menu. The File menu has only the Print, Print Preview, Print Setup, and Exit commands. If you had chosen "Both a database view and file support," AppWizard would have supplied the missing File menu commands.

▶ To view the Enroll menu resource

1. In ResourceView, expand the Enroll.rc folder.

   This displays the resources associated with a project.

2. Expand the Menu folder.

3. Double-click IDR_MAINFRAME.

   The Menu Editor opens, displaying the default menu that AppWizard created for the Enroll application.

4. Click the File menu item to view its structure.

5. Close the Menu Editor when you're finished.

Note If you choose the option "Both a database view and file support" in AppWizard, the document class plays two roles. First, it serves as a proxy for the database. Second, it represents the file that is opened and saved via the New, Open, Save, and Save As commands on the File menu. This file might be used for a variety of purposes; for ideas, see "MFC: Using Database Classes Without Documents and Views" and "Serialization: Serialization vs. Database Input/Output" in *Programming with MFC*.

For more information about documents and views, see Chapter 3, "Working with Frame Windows, Documents, and Views" and Chapter 4, "Working with Dialog Boxes, Controls, and Control Bars" in *Programming with MFC*.

Customizing the Dialog Template for the Section Form

Along with the classes, AppWizard creates a dialog template resource named `IDD_ENROLL_FORM`, which the **CRecordView**-derived class, `CSectionForm`, uses to display its form controls. Because **CRecordView** is derived from **CFormView**, a record view's client area is laid out by a dialog template resource. The layout of the form is up to you. AppWizard places one static text control on the dialog template resource, labeled "TODO: Place form controls on this dialog."

In the following procedure, you'll replace this text with controls that correspond to columns in the table (via the field data members of the recordset).

▶ To customize Enroll's form

1 In ResourceView, expand the Enroll Resources folder.

2 Expand the Dialog folder.

3 Double-click IDD_ENROLL_FORM.

The dialog editor opens and displays the dialog box with the corresponding ID.

For more information about the dialog editor, see Chapter 6, "Using the Dialog Editor," in the *Visual C++ User's Guide*.

4 Select, and then delete the static control that says "TODO: Place form controls on this dialog."

5 Design Enroll's Section form to resemble Figure 31.3, using static controls and edit controls.

Tip You can press CTRL before you click a dialog box control; then release the CTRL key and click in the dialog box multiple times to add multiple copies of the control. For instance, if you want six edit controls, click six times. Click the selection arrow to stop adding controls.

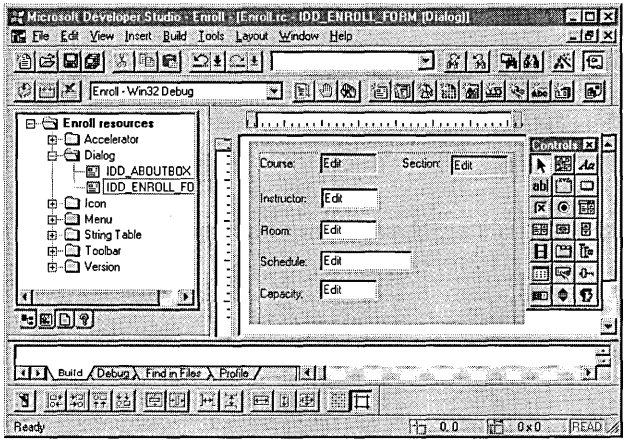
Resize the dialog as needed. You can either add the controls in pairs (that is, first a static text control and then the corresponding edit control, and so on) or later change the Tab order of the controls so that they are paired in this way. This becomes important when you bind the controls to recordset fields.

6 Choose Properties from the Edit menu to display the Properties window, and then pin it down so that it stays open while you add and edit the dialog box controls.

7 For each edit control, use the ID box in the Properties window to specify an ID based on the table column names (for example, `IDC_COURSE`). This is only a convention, but it is used throughout the tutorial.

Note The "Edit" caption that appears in each edit control is not visible to the user at run time, so you needn't worry about deleting it. To view the dialog as it will appear at run-time, press CTRL+T to enter test mode for the dialog. Press ESC to cancel test mode.

Figure 31.3 The Layout of Enroll's Section Form



8 Make the Course and Section edit controls read-only. To do so, select the Styles page in the Properties window and set the Read Only check box. (The other edit controls are updatable.)

   According to a common rule in the user-interface design of database forms, the user shouldn't be able to update these key fields. If users want to change a course number or section of a Section record, they must delete the old Section record and add a new one to avoid possibly violating the referential integrity of the database. Enroll tutorial Step 3 implements Add and Delete functionality.

9 Save ENROLL.RC.

   It's a good idea to periodically back up your work.

If you did not add the static text and corresponding edit controls in order, one after the other, you need to change the tab order. In either case, you can easily check the current tab order and change it if necessary.

▶ To view or change the tab order of controls

1 With the dialog resource open, from the Layout menu choose Tab Order.

   You'll see numbers depicting the current tab order of the controls.

2 Specify the tab order you want by clicking each control in that order.

   As you click, you'll see the numbering change to reflect your choice.

   For Enroll, specify a tab order such that each edit control is preceded in the tab order by the static text control that describes it. By specifying this tab order, you enable ClassWizard to derive a name for the edit control when you bind it to a data member, as you'll do in the next section.

3 Press ESC to exit Tab Order mode.

Binding Enroll's Controls to Recordset Fields

With the form designed, it's time to indicate which edit controls map to which table columns—or, more precisely, which controls map to which recordset field data members. To perform this task, you use ClassWizard's "foreign object" mechanism. (For details about these foreign objects, see the article "ClassWizard: Foreign Objects" in *Programming with MFC.*)

Normally, you use ClassWizard to bind controls in a dialog box or form to member variables of your **CDialog**- or **CFormView**-derived class. In the case of **CRecordView**, though, you bind the form's controls not to data members of the record view class but to data members of the recordset class associated with the record view.

Your **CRecordView**-derived class—`CSectionForm` in this case—has a data member called `m_pSet`. You can view `m_pSet` in the ClassView of `CSectionForm`. This data member is a pointer to `CSectionSet`, Enroll's recordset class. Recall that you viewed this recordset class on the Member Variables tab of ClassWizard (see Figure 31.2).

The control bindings go through `m_pSet` to the corresponding field data members of `CSectionSet`. For example, in the following procedure, you will bind the Course edit control to:

```
m_pSet->m_CourseID
```

▶ To bind a form control to a recordset data member

1. If necessary, choose `IDD_ENROLL_FORM` from the Dialog folder in ResourceView, to open the dialog box inside the dialog editor.

   For more information about the dialog editor, see Chapter 6, "Using the Dialog Editor," in the *Visual C++ User's Guide.*

2. In the dialog editor window, hold down the CTRL key and double-click the Course edit control.

   ClassWizard's Add Member Variable dialog box appears, with a proposed field name selected for you in the Member Variable Name box. ClassWizard chooses this name based on the caption of a static text control that falls previous to the edit control in the tab order.

   For example, for `IDC_COURSE`, the control's caption is "Course," and the Member Variable Name box should display:

   ```
   m_pSet->m_CourseID
   ```

3. Click OK in the Add Member Variable dialog box to accept the name.

4. Repeat steps 2 and 3 for each of the other edit controls on the form.

It isn't necessary to create mappings for the static text controls.

5 Save your work.

Note Using CTRL+double-click in the dialog editor is a ClassWizard shortcut for mapping form controls to members of the associated dialog, form view, or record view class. Use it on a pushbutton to create a command handler function for the button. Use it on other controls to create a class member variable.

You can view the complete mappings in the Class Wizard Member Variables tab for class `CSectionForm`. For example, where `IDC_COURSE` appears in the Control IDs column, you'll see `->m_CourseID` in the corresponding Member column.

Build and Run Enroll Step 1

Build and run Enroll Step 1. For information on building, see "Build the Starter Application," in Chapter 3.

When the `CSectionSet` recordset opens, it selects records from the Section table in the Student Registration database. The first record becomes the "current record" in the recordset. Enroll's database form displays the controls you designed, now filled with data from the current record.

Try a few things:

- Take a look at the Record menu, which has First Record, Previous Record, Next Record, and Last Record commands. (The toolbar has buttons that correspond to the menu commands.) Try using the commands to scroll through the records in the recordset.
- Update some of the fields. The new values are accepted into the data source when you move to another record. As mentioned earlier, the key fields Course and Section are read-only.

  Note In order to make changes to the fields, you must have write access to the data source.

This completes Step 1 of the database tutorial. Chapter 32 continues by showing you how to add a second recordset and use it to fill a combo box control on the form.

CHAPTER 32

Using a Second Recordset

Although AppWizard starts you off with one initial pair of recordset and record view classes, you can later use ClassWizard to add more recordset and record view classes. Multiple record views can view the same recordset. Conversely, a record view class can view more than one recordset, although only one of the recordsets can be its primary recordset.

In this chapter, you'll add some code so that you can view more than one recordset with the same record view.

This chapter explains:

- Replacing the Course edit box with a Course List combo box.
- Creating a second recordset class with ClassWizard.
- Filling the Course List combo box from the second recordset.
- Parameterizing the Section recordset.
- Requerying the Section recordset.
- Building and running Enroll Step 2.

If you choose to work along with the tutorial, perform all the steps in the procedures in this chapter. At the end, you'll be able to build and run the Enroll Step 2 application.

About Step 2

Step 2 teaches:

- Using more than one recordset in the same record view.
- Filling a combo box from a recordset.
- Using a recordset filter (**CRecordset::m_strFilter**).
- Sorting a recordset (**CRecordset::m_strSort**).

- Using recordset parameters.
- Refreshing a recordset by calling the **CRecordset::Requery** member function—a common task if you use filters and parameters.

Step 2 illustrates using two recordsets in one record view by implementing a second recordset for the Course table, which is used to fill a combo box in the `CSectionForm` view. In this way, the `CSectionForm` view has a primary association with the `CSectionSet` recordset—the form shows one record from `CSectionSet`—while the combo box is associated with a second recordset, `CCourseSet`.

This step changes the `CSectionSet` recordset so it selects only the available class sections for a single course name, rather than selecting all class sections for all courses. You'll change the Course edit control to a combo box control and fill the combo box with all of the course names from the Course table. When the user selects a different course name from the combo box, you'll requery the Section table to select only those class sections for the course name the user chose.

Trying Out Step 2

Try Step 2 out now, if you like, by running it from Books Online. (For information about how to do this, see "Previewing the Sample Applications.")

Enroll's database form now displays a list of course names in the combo box. The other controls are filled from the first Section record for that course name.

Use the First Record, Next Record, Previous Record, and Last Record commands on the Record menu (or the equivalent toolbar buttons) to move through the different class sections for the same course name.

Note The `CSectionForm` record view detects the end of a recordset only if the user has moved past it. The user must move beyond the last record before the record view can tell that it must disable any user-interface objects for moving to the next or last record.

Select a new course name from the Course combo box. The application then requeries the `CSectionSet` recordset for the new course name. Move through the class sections for the new course name.

Exit Enroll Step 2 when you finish exploring.

Changing the Course Control to a Combo Box

The Course edit control started out as an edit control in Step 1. In Step 2, you will use the dialog editor to change it to a drop-list style combo box. For more information about using the dialog editor, see Chapter 6, "Using the Dialog Editor," in the *Visual C++ User's Guide.*

To perform the procedures in Step 2, start with the version of ENROLL you created in Step 1 of the Enroll Tutorial.

▶ To change the Course control to a combo box

1 Open the dialog resource whose ID is `IDD_ENROLL_FORM`.

2 Select, and then delete the Course edit control.

3 Add a combo box where the edit control was.

4 Double-click the combo box control to open the Properties window, pin it down, and then specify the following:

- In the ID box on the General property page, type `IDC_COURSELIST`.
- From the Type list box on the Styles property page, choose Drop List.

5 Increase the size of the combo drop-down box so it can show more than two course names at a time:

- Click the drop-down arrow on the right side of the combo box.
- Use the bottom sizing handle to extend the drop-down area downward enough to hold several lines of text.

In the next step you'll change the tab order of the dialog to make sure that the new control follows its associated static text label in the tab order. This enables ClassWizard to match the recordset member to the static text caption.

6 From the Layout menu, choose Tab Order.

7 With the tab order showing, click the Course static text control, and then click the Course combo box control.

The tab order should ripple through the other controls, keeping the sequential relationship between each static text and its associated edit control.

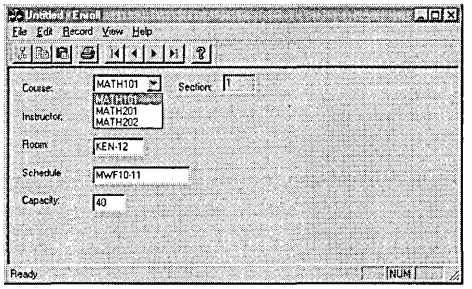
See Chapter 6, "Using the Dialog Editor," in the *Visual C++ User's Guide* for information about setting the tab order. For general information about ClassWizard, see the *Visual C++ User's Guide*.

8 Press ESC, or click in the dialog box to exit Tab Order mode.

Leave the dialog editor open for the next procedure.

Figure 32.1 shows the final appearance of Enroll Step 2 with the combo box in place.

Figure 32.1 Enroll Step 2 With a Combo Box



Binding the Combo Box Control to a Recordset Field and a CComboBox Variable

Now that you've replaced the edit control for the `m_CourseID` member of `CSectionSet` with a combo box, you need to:

- Unbind the old edit control.
- Bind the new combo box control to the CourseID field.
- Bind the combo box control to a second member variable, a **CComboBox** variable in `CSectionForm`.

You will later use member functions of **CComboBox**, such as **AddString**, to fill and read the combo box.

When you complete the next three procedures, the combo box control will have two member variables associated with it: (1) the foreign member variable, `m_CourseID`, in the recordset associated with the record view, and (2) the **CComboBox** member variable in the record view class.

ClassWizard supports having two such member variables bound to the same control.

▶ To remove the old edit control binding

1 From the View menu, choose ClassWizard.

2 Choose the Member Variables tab.

3 In the Class Name box, select class CSectionForm (if it isn't already selected).

4 In the Control IDs box, select IDC_COURSE and choose Delete Variable.

5 Click OK.

▶ To bind the combo box control to the recordset member

1 In the dialog editor, press CTRL and double-click the combo box control to open the ClassWizard Add Variable Member dialog box.

Recall that when you changed the edit control to a combo box, you also changed the tab order so the combo box would directly follow the Course static text control. ClassWizard chooses the member variable name based on the recordset member associated with the static text control that directly precedes it in tab order.

2 Click OK to accept `m_pSet->m_CourseID` as the member variable name, and to exit the Add Variable member dialog box.

▶ To bind the combo box control to the view's CComboBox member variable

1 Open ClassWizard and if necessary, select the Member Variables tab.

2 In the Control IDs list, highlight IDC_COURSELIST, and choose Add Variable.

3 In the Member Variable Name box, type `m_ctlCourseList`.

4 In the Category box, select Control.

Note that this automatically selects **CComboBox** as the Variable Type.

5 Click OK to exit the Add Member Variable dialog box.

6 Click OK to exit ClassWizard.

7 Save your work.

You'll probably want to close the dialog editor and property page.

Creating a Recordset for the Course Table

Enroll already has one recordset, for the Section table, which fills the controls on the `CSectionForm` record view with information about a single class section of the currently selected course name. Now you'll add a second recordset, for the Course table, used to fill the combo box control with a list of all available course names.

▶ To create a new recordset class

1 From the View menu, choose ClassWizard.

2 Select the Add Class menu button, and from the menu select New.

This opens the Create New Class dialog.

3 In the Name box under Class Information, type `CCourseSet`.

4 From the Base Class drop-list, select CRecordset.

5 Clear the Add to Component Gallery checkbox.

For more information about this option, see "Using Component Gallery" in Chapter 15 of the *Visual C++ User's Guide*.

6 Click the Create button.

This opens the Database Options dialog box.

▶ To connect the recordset class to the Course table

1 From the ODBC drop-list, select Student Registration, and click OK.

Depending on the database type, you may need to supply additional information to log in to the data source.

The Select DatabaseTables dialog box opens.

2 Select the table name COURSE, and choose OK. Depending on the data source type you are using, additional qualifiers may precede or follow the table name.

This connects the table name to class `CCourseSet` and returns you to the ClassWizard Member Variables tab. The Class Name box shows `CCourseSet`, and three names are listed in the Column Names box.

Table 32.1 shows the column names, their data members, and their data types.

3 Choose OK to close ClassWizard.

Note On the Member Variables tab, you can see that all of the table's columns are already assigned to field member variables. You can use ClassWizard to delete those variables if you don't need to access or modify the columns—but be careful not to delete a field member variable for a column that is part of the table's primary key.

Table 32.1 CCourseSet Data Members

Column name	Type	Data member
CourseID	**CString**	`m_CourseID`
CourseTitle	**CString**	`m_CourseTitle`
Hours	**int**	`m_Hours`

For more information about using ClassWizard to create recordset classes, see the article "ClassWizard: Creating a Recordset Class" in *Programming with MFC*.

Embedding the Recordset Object in the Document Object

In Step 1, AppWizard embedded the `CSectionSet` object in the document. In this step, you'll do the same for the second recordset object—an object of the `CCourseSet` class that you created earlier with ClassWizard.

▶ To embed the recordset in the document

1. From FileView, double-click file EnrollDoc.h to open it.

   Header files can be found under the Dependencies folder

2. Declare an embedded `CCourseSet` object, by adding the following line to the public Attributes section, just beneath the `CSectionSet` declaration:

   ```
   CCourseSet  m_courseSet;
   ```

3. Similarly, open files EnrollDoc.cpp, Enroll.cpp, and SectionForm.cpp, and add a **#include** directive for "CourseSet.h" before the existing **#include** directive for "EnrollDoc.h", as shown in the following line:

   ```
   #include "CourseSet.h"
   ```

4. Save EnrollDoc.h and the .cpp files.

The document's `m_courseSet` member is referred to in the implementation of `OnInitialUpdate` that you'll complete later.

Filling the Combo Box with a List of Courses

A good place to fill the combo box with a list of course names is in `CSectionForm`'s override of **CRecordView**'s **OnInitialUpdate** member function. As part of its own initialization, the form fills the combo box. The overall logic is as follows:

1. Construct and open a `CCourseSet` recordset based on the Course table.
2. Remove any current entries in the combo box.
3. For each course name in `CCourseSet`, add the CourseID to the combo box.
4. Set the selection to the first course name (as sorted) in the combo box.

The code in the following procedure fills the combo box and also filters, parameterizes, and sorts the `CSectionSet` recordset. Filtering, parameterization, and sorting are explained in sections that follow.

▶ To fill the combo box

1 From the Window menu, select the file SectionForm.cpp. (This file should still be open from the previous procedure.)

2 Use ClassView or WizardBar to navigate to the implementation of `OnInitialUpdate`.

3 Just after the first line — `m_pSet = &GetDocument()->m_sectionSet;` — add the code below (don't replace any code):

```
// Fill the combo box with all of the courses
CEnrollDoc* pDoc = GetDocument();
pDoc->m_courseSet.m_strSort = "CourseID";
if (!pDoc->m_courseSet.Open())
    return;

// Filter, parameterize and sort the CSectionSet recordset
m_pSet->m_strFilter = "CourseID = ?";
m_pSet->m_strCourseIDParam = pDoc->m_courseSet.m_CourseID;
m_pSet->m_strSort = "SectionNo";
m_pSet->m_pDatabase = pDoc->m_courseSet.m_pDatabase;
```

You'll add the necessary member declaration in the section "Setting Up the Parameter."

4 Next, after the last line (`CRecordView::OnInitialUpdate();`), add the following code:

```
m_ctlCourseList.ResetContent();
if (pDoc->m_courseSet.IsOpen())
{
    while (!pDoc->m_courseSet.IsEOF())
    {
        m_ctlCourseList.AddString(
            pDoc->m_courseSet.m_CourseID);
        pDoc->m_courseSet.MoveNext();
    }
}
m_ctlCourseList.SetCurSel(0);
```

5 Save your work.

For more information, see the article "Record Views: Filling a List Box from a Second Recordset" in *Programming with MFC*.

Filtering and Parameterizing the Recordset

The Step 1 version of Enroll selects into `CSectionSet` all of the records in the Section table. In Step 2, only the class sections for a specific course name should be selected. This discussion introduces the concepts of recordset filters and parameters.

Setting Up the Filter

Note You've already added the code to filter and parameterize the `CSectionSet` recordset (in `OnInitialUpdate`); the code in this section is for illustrative purposes only. Do not add the code from this section to your source files.

A recordset filter determines what subset of records is selected from a table or query. To add a filter, you simply set the value of **CRecordset::m_strFilter** before calling **CRecordset::Open**. For example, the following code selects just the class section records for course MATH101:

```
m_pSet->m_strFilter = "CourseID = 'MATH101'";
m_pSet->Open();
```

Since the base class **CRecordView::OnInitialUpdate** calls **CRecordset::Open**, all you need to do to initially select the records for MATH101, for example, is replace the following AppWizard implementation of `OnInitialUpdate`:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;
    CRecordView::OnInitialUpdate();
}
```

with:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;
    m_pSet->m_strFilter = "CourseID = 'MATH101'";
    CRecordView::OnInitialUpdate();
}
```

The filter can be any logical expression that is legal for the SQL **WHERE** clause. For example, the following is legal:

```
m_pSet->m_strFilter =
                "CourseID = 'MATH101' AND InstructorID = 'ROGERSN'";
```

Examine the `OnInitialUpdate` code you added earlier. It shows the filter for `CSectionSet` in Enroll Step 2.

Caution In Enroll, filter strings typically use a parameter placeholder, "?", rather than assigning a specific literal value, such as "MATH101", at compile time. If you do use literal strings in your filters (or other parts of the SQL statement), you may have to "quote" such strings with a DBMS-specific "literal prefix" and "literal suffix" character(s). For example, the code in this section uses a single quote character to bracket the value assigned as the filter, "MATH101". You may also encounter special syntactic requirements for operations such as outer joins, depending on your DBMS. Use ODBC functions to obtain this information from your driver for the DBMS. For example, call **::SQLGetTypeInfo** for a particular data type, such as **SQL_VARCHAR**, to request the **LITERAL_PREFIX** and **LITERAL_SUFFIX** characters.

If you're writing database-independent code, see Appendix C in the *ODBC Programmer's Reference* for detailed syntax information.

Setting Up the Parameter

Enroll reselects, or "requeries," class section records every time the user selects a new course name from the combo box. One way to implement this is to close the old `CSectionSet` object and reopen it by supplying a new **m_strFilter** value before calling **Open**. This works but is somewhat inefficient, because the framework has to completely reconstruct and invoke a new SQL **SELECT** statement. A more efficient way to requery the same recordset is to "parameterize" the filter—call **Requery** with a new filter value and a specific parameter value.

In order to parameterize the filter, you'll perform the following procedures:

- Declare a parameter data member in the recordset's header file.
- Bind the parameter data member to the recordset.

To implement the **Requery** with a new filter and a specific parameter value supplied at run time, you:

- Specify a parameterized filter.
- Supply the run-time parameter value.

The following sections describe these procedures.

▶ To declare a parameter data member in the recordset's header file

1 Open file SectionSet.h.

2 Add the following member variable declaration for `m_strCourseIDParam`, just before the //Overrides section, after the `//}}AFX_FIELD` line:

```
CString m_strCourseIDParam;
```

▶ To bind the parameter data member to the recordset

1 Use ClassView to navigate to the `CSectionSet` constructor, and initialize the parameter count variable, **m_nParams**, which by default is zero. Also initialize Enroll's single parameter, `m_strCourseIDParam`.

Place the following two lines of code after the line `//}}AFX_FIELD_INIT`:

```
m_nParams = 1;
m_strCourseIDParam = "";
```

2 Use ClassView or WizardBar to navigate to the `DoFieldExchange` member function definition, and add the following two lines of code to identify `m_strCourseIDParam` as a parameter data member. Add the code at the end of the function, after the `//}}AFX_FIELD_MAP` line.

```
pFX->SetFieldType(CFieldExchange::param);
RFX_Text(pFX, "CourseIDParam", m_strCourseIDParam);
```

`DoFieldExchange` recognizes two kinds of fields: columns and parameters. The call to the **CFieldExchange** member function **SetFieldType** indicates what kind of field(s) follow in the RFX function calls. In this example, there is one parameter: `m_strCourseIDParam`.

The name of the column for the parameter in the **RFX_Text** call—"CourseIDParam"—is arbitrary; you can provide any name you want.

3 Save your work.

▶ To specify a parameterized filter

- Before the call to the base class function **CRecordset::Open**, which is called by **CRecordView::OnInitialUpdate**, specify the parameterized filter, as shown in this line (which you've already added):

```
m_pSet->m_strFilter = "CourseID = ?";
```

The question mark "?" indicates where the parameter value will be substituted at run time. If you have more than one parameter in your **m_strFilter**, such as:

```
m_pSet->m_strFilter = "CourseID = ? AND SectionNo = ?";
```

you must make multiple RFX calls after the call to:

```
pFX->SetFieldType(CFieldExchange::param);
```

You must make the RFX calls for multiple parameters in exactly the same order as the question marks in the **m_strFilter** and/or **m_strSort**.

Note If you have both a filter and a sort with parameters, specify the filter parameters first, then the sort parameters. Not all ODBC drivers permit parameters on a sort. Consult the Help file for your ODBC driver.

▶ To supply the run-time parameter value

- Assign the value to the previously bound parameter data member, as shown in the following line (which you've already added in the **OnInitialUpdate** function).

```
m_pSet->m_strCourseIDParam = pDoc->m_courseSet.m_CourseID;
```

This sets the parameter value to be the first course record retrieved from the `CCourseSet` recordset. All parameter values must be assigned before calling **CRecordset::Open** (or **CRecordView::OnInitialUpdate**) or, as you will see later, before calling **CRecordset::Requery**.

Reusing a Database Object Opened by Another Recordset

Note You add no new code to Enroll in this section.

AppWizard and ClassWizard both implement **CRecordset**-derived classes such that the recordset object owns its own **CDatabase** object. Up to now, the **CDatabase** object has been transparent because the framework created it for you when you constructed a recordset object. The default implementation of **CRecordView::OnInitialUpdate** indirectly calls the wizard-implemented `GetDefaultConnect` function for the recordset. The implementation looks like this:

```
CString CSectionSet::GetDefaultConnect()
{
    return "ODBC;DSN=Student Registration;";
}
```

The framework passes this "connection" string to **CDatabase::Open** for the **CDatabase** object that the framework creates in its implementation of **CRecordset::Open**. If your application has two or more recordsets, each recordset will, by default, create and open its own **CDatabase** object. If multiple recordsets access the same data source, it's a good idea to have them share the same **CDatabase** object.

One way to share the same **CDatabase** object among multiple recordsets is to pass the **m_pDatabase** member of the first recordset object to the **Open** function of the other recordsets. This is what you've already implemented in `CSectionForm::OnInitialUpdate`:

```
m_pSet->m_pDatabase = pDoc->m_courseSet.m_pDatabase;
CRecordView::OnInitialUpdate();
```

If **CRecordset::Open** finds that the **m_pDatabase** member is already allocated, it simply reuses the open **CDatabase**.

Another way to share the same **CDatabase** object among multiple recordsets is to embed the **CDatabase** object in the document object. For an example of this approach, see the source code for Enroll Step 4 in Samples \ MFC Samples \ Tutorials in Books Online.

Sorting the Recordset

The procedure for sorting a recordset is very simple: set the member variable **CRecordset::m_strSort** before calling **CRecordset::Open**. The syntax for **m_strSort** is exactly that of the SQL **ORDER BY** clause, which is one or more columns separated by commas.

The `CCourseSet` records are all sorted by CourseID (which you have already added):

```
pDoc->m_courseSet.m_strSort = "CourseID";
```

Also, the `CSectionSet` records for a given course name are sorted by class section:

```
m_pSet->m_strSort = "SectionNo";
```

For more information about using SQL with the database classes, see the article "SQL" in *Programming with MFC*.

Requerying the CSectionSet Recordset

Whenever the user selects a new course name from the combo box, Enroll must "requery" the `CSectionSet` recordset to refresh its records. By selecting a new course name, the user will see records only for the class sections of that course name. The existing `CSectionSet` recordset contains records for the previous course name. Requerying the recordset brings it up to date for the new course name, using the current values of the filter and sort strings.

When the user accepts a selection in the combo box, the `CSectionForm` record view gets a **CBN_SELENDOK** notification message. The record view uses its handler for this message to reselect records based on the course name selected, passing the course ID as a parameter.

The following procedure describes how to use WizardBar to create this handler. For more information, see "Using WizardBar" in the *Visual C++ User's Guide*.

▶ To requery the CSectionSet recordset

1 From the Window menu, select SectionForm.cpp.

2 In the WizardBar Object IDs drop-list, select IDC_COURSELIST.

3 In the Messages drop-list, select CBN_SELENDOK.

4 Respond Yes when prompted to create a handler.

5 In the editor window, add the following code in place of the //TODO comment:

```
if (!m_pSet->IsOpen() )
    return;
m_ctlCourseList.GetLBText(m_ctlCourseList.GetCurSel(),
    m_pSet->m_strCourseIDParam);
m_pSet->Requery();
if (m_pSet->IsEOF())
{
    m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
    m_pSet->m_CourseID = m_pSet->m_strCourseIDParam;
}
UpdateData(FALSE);
```

6 Save your work.

This code requeries records from the database into the recordset, based on the parameter value in `m_strCourseIDParam`. The parameter value is set to the currently selected course name from the Course List combo box before requerying the database.

If you requery and it turns out that the selected course name has no class sections, the recordset is initialized with Null database field values except for CourseID.

For more information, see "Recordset: Requerying a Recordset (ODBC)" in *Programming with MFC*.

Build and Run Enroll Step 2

If you're working along, build and run your version of Enroll Step 2. Use the navigation user interface to move through all class sections for the course name currently selected in the Course combo box. Select a different course name in the combo box and navigate through its class sections.

This completes Step 2 of the database tutorial. Chapter 33, "Adding and Deleting Records," (Step 3) concludes the tutorial by showing you how to add and delete records.

CHAPTER 33

Adding and Deleting Records

This tutorial step implements new commands for adding and deleting records and for abandoning an update in progress. This chapter explains:

- Creating the Step 3 user interface.
- Adding, editing, and deleting records.
- Implementing the Add, Refresh, and Delete commands.
- Building and running Enroll Step 3.

If you choose to work along with the tutorial, perform all the steps in the procedures in this chapter. At the end, you'll be able to build and run your Enroll Step 3 application.

About Step 3

Step 3 teaches:

- The basics of adding, editing, and deleting records.
- Implementing commands for these operations.

Up to now Enroll has supported editing (updating) records but not adding or deleting records.

There are many different user-interface styles for adding records. For example, when a Microsoft Access user reaches the end of a recordset, Access considers the next record to be a new record. Other applications have an explicit Add command. Enroll's user interface is only one among many possible user interfaces that you might implement using MFC.

The user interface in Step 3 includes three new commands on the Record menu, with corresponding toolbar buttons:

- The Add command prepares a blank record into which the user enters data. The user saves the new record by moving to another record, just as he or she saves an

edited record by moving to another record. The user can also save the new record by issuing the Add command again.

- The Refresh command abandons an operation to add or edit a record. Refresh restores the modified record to its original state or returns to the record shown before Add.
- The Delete command deletes a record.

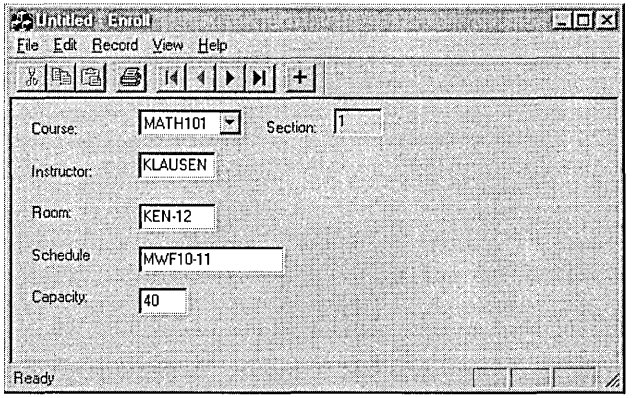
Try Step 3 out now, if you like. Run Enroll.exe from Books Online. For more information, see "Previewing the Sample Applications."

Here are some things to try:

- Try the new Add, Refresh, and Delete commands.
- Try forcing the two exceptions handled by Enroll:
  - Try to delete a section that has Enrollment records.
  - Try to add a duplicate section.

When you finish, exit the program. Figure 33.1 shows the finished Enroll application.

Figure 33.1 The Enroll Step 3 Application



Creating the Step 3 User Interface

In this section you'll use the menu editor to add menu commands to Enroll's default menu. In later sections you'll add (optional) accelerators and message handler member functions for these commands.

For more information about editing menu resources, see Chapter 6, "Using the Menu Editor," in the *Visual C++ User's Guide.*

Add Menu Items for Add, Refresh, and Delete

In the following procedure, you'll add three new menu items to Enroll's Record menu: Add, Refresh and Delete. The command IDs you assign to the new menu items are application-specific IDs, not predefined by the framework as are **ID_RECORD_FIRST** and the other commands on the Record menu.

Tip You may find it helpful to "pin down" the Menu Item Properties dialog during the following procedure.

▶ To add menu items for the commands

1 In ResourceView, expand the Enroll resources folder if necessary, and then expand the Menu folder.

2 Open the **IDR_MAINFRAME** menu resource, and open the Menu Item Properties page by choosing Properties from the Edit menu.

3 At the top of the existing Record menu items, add an "Add" menu item with the following caption, resource ID, and command prompt:

- &Add
- ID_RECORD_ADD
- Add a new section

4 Add a "Refresh" menu item with the following caption, resource ID, and command prompt:

- &Refresh \tEsc
- ID_RECORD_REFRESH
- Cancel changes on form, or cancel Add

The "\t Esc" coding specifies that the ESC key can be used as an accelerator. If you do not complete the next procedure, "Add an Accelerator for the Refresh Command," don't include this code.

5 Add a "Delete" menu item with the following caption, resource ID, and command prompt:

- &Delete
- ID_RECORD_DELETE
- Delete section

6 Add a separator:

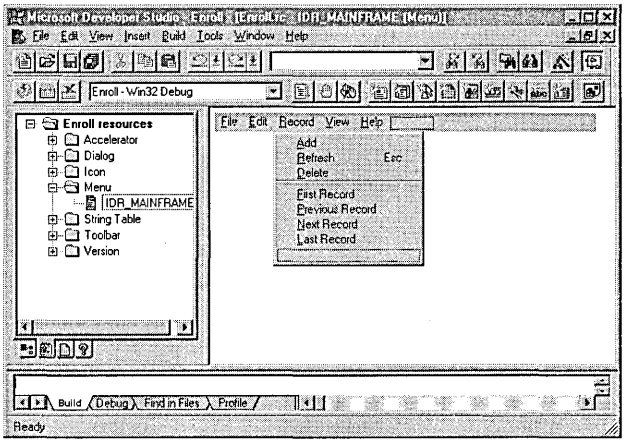
- Insert a new menu item and, in the Menu Item Properties dialog, select the Separator checkbox.

7 Save your work and leave the menu editor window open.

You'll need the editor open to establish a context when you use ClassWizard to create command handler functions for the menu commands you've just added.

Figure 33.2 shows the completed menu in the menu editor.

Figure 33.2 The Record Menu with New Commands



Add an Accelerator for the Refresh Command

You can skip this step if you wish, since you can test the application without this accelerator.

For information about creating and editing accelerators, see Chapter 8, "Using the Accelerator Editor," in the *Visual C++ User's Guide*.

▶ To add an accelerator

1. In ResourceView, expand the Accelerator folder.
2. Open the **IDR_MAINFRAME** accelerator resource.

   Note The name for this resource doesn't need to match the menu resource name, so long as the ID you assign to an accelerator matches the ID for the corresponding menu item.
3. Create a new accelerator with the following ID: ID_RECORD_REFRESH.
4. From the Key drop-down list, choose VK_ESCAPE (or type it in).
5. Clear the "Ctrl" modifier box.
6. Save your work but leave the accelerator editor open.

   You'll need this editor or the menu editor open to establish a context when you use ClassWizard to create command handler functions for the menu commands.

Create Handlers for Add, Refresh, and Delete

Each of the new Record menu commands needs a command handler function in the `CSectionForm` class. Since the Enroll menu resource is associated with the `CMainFrame` class, you need to make an association between the menu IDs and the `CSectionForm` class. You do this by giving focus to the **IDR_MAINFRAME** resource (accelerator or menu) so ClassWizard can glean the available command IDs from the resource.

▶ To create handlers for the commands

1. With focus on the **IDR_MAINFRAME** resource, open ClassWizard and choose the Message Maps tab.

   Notice that the `CMainFrame` class is automatically selected.

2. In the Class Name box, select CSectionForm.
3. In the Object IDs list, select the ID_RECORD_ADD command ID; in the Messages box, select COMMAND; and then choose Add Function to create a command handler function.

   Accept the default handler name: OnRecordAdd.

4. Repeat step 3 for the ID_RECORD_DELETE and ID_RECORD_REFRESH command IDs.
5. Click OK to exit ClassWizard.

   You can also close the resource editors at this point.

You'll fill in the command handlers in later sections.

The Basics of Adding, Editing, and Deleting Records

Before you implement the new command handlers, you should know some basic facts about how the framework supports database updating:

- **CRecordView** automatically updates the current record when the user moves to another record.
- **CRecordView** takes three steps to modify an edited record in the associated recordset when the user moves to another record. The record view:
  - Prepares the current record for updating by calling the recordset's **Edit** member function.
  - Calls the **UpdateData** member function derived from **CFormView**, which changes the recordset's member variables, usually by getting the new values from the form's controls.

- Calls the recordset's **Update** member function to actually update the data source with the modified values.

- **CRecordView** does not provide a default implementation for Add, since user interfaces for Add functionality vary widely among database applications.
- The steps for adding a new record parallel the steps for updating a modified record:
  - Prepare a new record by calling the recordset's **AddNew** member function. The fields of the new record are initially Null. (In database terminology, Null means "having no value" and is not the same as **NULL** in C++.)
  - Change the recordset's member variables, usually by getting the new values from the form's controls with **UpdateData**.
  - Call the recordset's **Update** member function to actually update the data source with the values for the new record.
- Deleting a record is simpler than adding or editing one. The record view simply calls the recordset's **Delete** member function.

  There are two main concerns when you delete a record. First, if you delete a record from one table and there are related records in other tables, you may damage the integrity of your database. For example, deleting a class section for which there are records in the Enrollment table makes the Section and Enrollment tables inconsistent.

  Second, after deleting a record, you or the user must move off the deleted record to another record.

Implementing the Add Command

Step 3 implements a user interface for Add that closely parallels **CRecordView**'s default user interface for modifying an existing record. The user starts a new record with the Add command on the Record menu.

Implementing the Command Handler

In response to the Add command, the record view calls its `OnRecordAdd` member function and enters an "add mode" by setting an `m_bAddMode` data member to **TRUE**. The add mode is completed when the user moves off the record. The Step 3 implementation overrides the record view's **OnMove** member function to implement completion of the add mode. The following procedures implement the add mode, and create a **CEdit** member variable used to turn on and off the read-only style of the Section edit control.

▶ To implement the Add mode

1 In the Attributes section of file SectionForm.h, add the protected `m_bAddMode` data member:

```
protected:
    BOOL m_bAddMode;
```

2 Initialize `m_bAddMode` in the `CSectionForm` constructor in file SectionForm.cpp. (You can jump directly to the constructor from ClassView.) Add the following line after the `//}}AFX_DATA_INIT` line:

```
    m_bAddMode = FALSE;
```

In Steps 1 and 2 of the tutorial, the Section control was read-only because it was necessary to prevent the user from changing this primary key value of the Section record. In Step 3, you need to turn off the read-only style of the Section control when the user is in add mode. The control is still read-only if the user is in browse/update mode rather than add mode.

To change the read-only style, you must call the **CEdit** member function **SetReadOnly** with the appropriate parameter. This requires a member variable of type **CEdit** in `CSectionForm`. At this point, the class has a **CString** data member representing the Section control, but you need a **CEdit** member variable as well.

▶ To define the CEdit member variable

1 Open ClassWizard and choose the Member Variables tab.

2 In the Class Name box, select CSectionForm.

3 In the Control IDs box, select IDC_SECTION, which is already associated with a **CString** member.

4 Choose Add Variable to open the Add Member Variable dialog box.

5 In the Member Variable Name box, type the name `m_ctlSection`.

6 In the Category box, select Control.

Notice that the Variable Type box changes appropriately to **CEdit**.

7 Choose OK to close the Add Member Variable dialog box.

Notice that a second member variable is now associated with the `IDC_SECTION` control ID. You access the control's value through `m_pSet->m_SectionNo`. You access the control itself, to call its member functions, through `m_ctlSection`.

8 Choose OK to close ClassWizard.

The Add command initiates add mode and calls the recordset's **AddNew** function to prepare a new record, but doesn't add the record to the data source. The record isn't actually added to the data source until a subsequent call to `OnMove` calls the recordset's **Update** function.

▶ To implement the OnRecordAdd command handler function

1 Use ClassView to jump to the `OnRecordAdd` skeleton handler that ClassWizard created in SectionForm.cpp.

2 Add the following code to implement the handler:

```
// If already in add mode, complete the previous new record
if (m_bAddMode)
    OnMove(ID_RECORD_FIRST);

CString strCurrentCourse = m_pSet->m_CourseID;
m_pSet->AddNew();
m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
m_pSet->m_CourseID = strCurrentCourse;
m_bAddMode = TRUE;
m_ctlSection.SetReadOnly(FALSE);
UpdateData(FALSE);
```

The most important line of this code is the call to **CRecordset::AddNew**, which prepares a new record. The rest of the code does the following:

- If the user is already in add mode, complete the current record by simulating the user's moving to another record. Moving to another record is the normal user interface for completing a record.
- Save the CourseID for the current record and use it as the default for the new record, based on the assumption that more often than not the user will want to add another section for the course currently being viewed.
- In add mode, change the Section control to read/write rather than read-only, so the user can enter a new section number.

Updating the Data Source with the Added Record

Add mode is completed when the user moves off the record. Enroll implements this by overriding the **CRecordView::OnMove** member function.

▶ To implement Add functionality in the OnMove function override

1 With SectionForm.cpp open in the text editor, select class CSectionForm in the WizardBar Object IDs drop-list.

2 In the Messages drop-list, select OnMove, and choose Yes when prompted to create a handler.

3 Fill in the skeleton `OnMove` function with the following code:

```
if (m_bAddMode)
{
    if (!UpdateData())
        return FALSE;
    TRY
    {
        m_pSet->Update();
    }
```

```
        CATCH(CDBException, e)
        {
            AfxMessageBox(e->m_strError);
            return FALSE;
        }
        END_CATCH

        m_pSet->Requery();
        UpdateData(FALSE);
        m_ctlSection.SetReadOnly(TRUE);
        m_bAddMode = FALSE;
        return TRUE;
    }
    else
    {
        return CRecordView::OnMove(nIDMoveCommand);
    }
```

In its default **CRecordView** implementation, **OnMove** moves to the next, previous, first, or last record. If the application has changed the recordset field data members for the current record before the move, the framework updates the data source before moving to another record.

Note Some ODBC drivers do not reflect newly added records in the recordset; others do. For those drivers that don't display newly added records, to make the added records visible you must requery the database. For more information, see "Recordset: Adding, Updating, and Deleting Records (ODBC)" in *Programming with MFC*.

Step 3 augments the default **CRecordView** user interface for updating the current record. If the user is in add mode and then moves off the new record, Enroll adds the newly prepared record to the data source before moving to another record. But you must decide whether it's important for added records to be immediately visible. For the tutorial, the decision is to requery the recordset after each add operation so the newly added record is included in the recordset.

Normally, the move commands behave as you might expect: Move Next moves to the next record, and so on. But as a consequence of the decision to requery during the add operation, when the user chooses any move command when adding a record, Enroll always effectively moves to the first record. That's because requerying the recordset automatically sets the recordset to the first record.

Disabling Combo Box Logic in Add Mode

Step 2 implemented a handler for selecting a course in the combo box. The handler requeried the parameterized `CSectionSet` for the newly selected course. In Step 3, the combo box takes on the additional duty of allowing the user to specify the course for a new section record being added. During add mode, you don't want to requery the recordset when the user selects a course from the combo box. Therefore, you need to put the requery logic inside an **if** clause that is executed only if add mode isn't in effect.

▶ To disable normal combo box logic while in add mode

1 Use ClassView to jump to the `OnSelendokCourseList` handler in class `CSectionForm`.

2 Place an **if** block around the requery code in the `OnSelendokCourseList` handler, so the handler now appears as follows:

```
void CSectionForm::OnSelendokCourselist()
{
    m_ctlCourseList.GetLBText(m_ctlCourseList.GetCurSel(),
        m_pSet->m_strCourseIDParam);
    if (!m_bAddMode)
    {
        m_pSet->Requery();
        if (m_pSet->IsEOF())
        {
            m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
            m_pSet->m_CourseID = m_pSet->m_strCourseIDParam;
        }
        UpdateData(FALSE);
    }
}
```

Implementing the Delete Command

In response to a Delete command, the record view deletes the current record by calling the **Delete** member function of its associated recordset.

▶ To implement the Delete command

1 Use ClassView to jump to the `OnRecordDelete` skeleton function in class `CSectionForm`.

2 Implement the handler with the following code:

```
TRY
{
    m_pSet->Delete();
}
CATCH(CDBException, e)
{
```

```
        AfxMessageBox(e->m_strError);
        return;
    }
    END_CATCH

    // Move to the next record after the one just deleted
    m_pSet->MoveNext();

    // If we moved off the end of file, move back to last record
    if (m_pSet->IsEOF())
        m_pSet->MoveLast();

    // If the recordset is now empty, clear the fields left over
    // from the deleted record
if (m_pSet->IsBOF())
        m_pSet->SetFieldNull(NULL);
    UpdateData(FALSE);
```

Catch any exceptions thrown by the recordset's **Delete** function so that errors are reported to the user. The **CDBException** data member **m_strError** is a fairly user-friendly error message, prepared by the underlying ODBC driver.

If you want to customize the error message, you can force the error condition, then examine **m_strStateNativeOrigin** for a particular state or native value. You can look up error messages in the ODBC *Programmer's Reference*, Appendix A, "ODBC Error Codes." Enroll takes the easy approach by displaying **m_strError**.

For Enroll, the decision was to move to the record following the deleted record. You could move to the previous record after a delete operation or anywhere else as long as you, or the user, moves off the deleted record.

Implementing the Refresh Command

The Refresh command cancels add mode, if the user had previously chosen Add, or it discards any changes the user may have made on the form for the current record. In the first case, Enroll cancels the add mode by calling:

```
CRecordset::Move(AFX_MOVE_REFRESH);
```

When you call **AddNew** to begin the add operation, the framework stores a copy of the current record's fields before allowing the user to enter new values in the record view's controls. Calling **Move** as shown here "refreshes" the current record—and effectively cancels the add operation. It restores the record that was current before add mode began. This also works if you called **Edit** instead of **AddNew**.

When the user cancels add mode, Enroll makes the Section control read-only again, for reasons explained earlier.

▶ To implement the Refresh command

1 Use ClassView to jump to the `OnRecordRefresh` skeleton handler in class `CSectionForm`.

2 Implement the handler function with the following code:

```
if (m_bAddMode)
{
    m_pSet->Move(AFX_MOVE_REFRESH);
    m_ctlSection.SetReadOnly(TRUE);
    m_bAddMode = FALSE;
}
// Copy fields from recordset to form, thus
// overwriting any changes the user may have made
// on the form
UpdateData(FALSE);
```

Note The source files for Enroll Step 3 on your distribution CD-ROM include functional toolbar buttons connected to the Add, Refresh, and Delete commands on the Record menu. The installed Step 3 source code supplies these toolbar buttons for Enroll. For more information on creating toolbar buttons, see "Edit Scribble's Toolbar" in Chapter 6, or "Using the Toolbar Editor" in Chapter 11 of the *Visual C++ User's Guide*.

Building and Running Enroll Step 3

Build and run your version of Enroll Step 3. Try the new Add, Refresh, and Delete commands. Try forcing the two exceptions handled by Enroll—try to delete a section that has Enrollment records, and try to add a duplicate section.

When you finish, exit the program.

This completes the database tutorial.

Note The sample source code for Enroll includes a fourth step, not covered in this tutorial. For a summary of what's included in Step 4, see "Enroll Step 4: A Preview."

CHAPTER 34

Data Access Objects (DAO) Tutorial

The DaoEnrol tutorial shows you how to develop an MFC database application using the Data Access Objects (DAO) database classes. This tutorial uses similar procedures and creates nearly the same application as the Enroll tutorial in Chapters 30 through 33.

In the DaoEnrol tutorial, you'll learn how to:

- Use AppWizard and ClassWizard for DAO database support.
- Create and use **CDaoRecordset** objects to open tables and run queries.
- Create and use **CDaoRecordView** objects for form-based applications.
- Use database support within the framework's document/view architecture.
- Add, update, and delete records.
- Manage multiple tables.
- Handle database exceptions.

This chapter contains:

- Instructions for setting up the Student Registration database for DAO.
- A brief overview of DAO.
- Step 1 of the DaoEnrol tutorial, which explains how to create a database application using AppWizard.

Once you have completed this chapter, you can go to Chapter 32 and follow the instructions found in the Enroll tutorial to complete the DaoEnrol tutorial.

There are some differences in the code generated for DaoEnrol and the code generated for Enroll. Consequently there are separate instructions for some sections of DaoEnrol that need to be substituted for sections in Chapters 32 and 33. These sections are included at the end of this chapter in "DaoEnrol Step 2" and "DaoEnrol Step 3."

Note The DAOENROL sample reflects the completed DaoEnrol tutorial, plus an undocumented fourth step. Separate source code is not provided for each of the DaoEnrol tutorial steps either under Samples in Books Online or in the \MSDEV\SAMPLES\MFC\DATABASE\DAOENROL subdirectory on the Visual C++ distribution CD.

Important This tutorial assumes you are familiar with Visual C++ and MFC. If you aren't, try the Scribble tutorial in Chapters 2 through 11 before you begin this tutorial. The Scribble tutorial introduces important class library concepts and techniques and teaches you to use the wizards and the resource editors.

It is not necessary to complete Enroll, the ODBC database classes tutorial, before starting DaoEnrol.

DaoEnrol and Enroll

DAO recordset, database, and record view class implementations are very similar to that of the ODBC database classes. This means that skills and knowledge you may already have as a result of using the ODBC database classes can apply to the DAO database classes. By completing both the Enroll and DaoEnrol tutorials, you have the opportunity to compare the code and study both implementations. You can also use DaoEnrol as a basis for an application that uses more of MFC's DAO functionality. For more information, see the articles "DAO: Writing a Database Application" and "Database Overview" in *Programming with MFC*.

The Tutorial Example: DaoEnrol

The tutorial example program, DaoEnrol, manages a student registration database similar to, but simpler than, a college registration system. It will help you to follow the tutorial if you understand the structure of the student registration database.

DaoEnrol is based on the same student registration database, STDREG32.MDB, that you use with the Enroll tutorial. However, it is not necessary to register this data source with the ODBC Administrator to use it with the DAO database classes in the DaoEnrol tutorial.

Table 34.1 lists the database tables, what they store, and the columns in them.

Table 34.1 Tables in the Student Registration Database

Table name	Contents	Column list
Course	Think of each record as an entry in a course catalog. Example: the MATH101 course.	CourseID* CourseTitle Hours
Section†	A section record is a specific offering of a course at a specific time. For example, MATH101 may have many sections.	SectionNo* CourseID* InstructorID Schedule RoomNo
Student	A record for each student at the school.	StudentID* Name GradYear
Enrollment	A record for each student in a particular section of a course. For a given student, there is an Enrollment record for each course the student is taking.	CourseID* SectionNo* StudentID* Grade
Instructor	A record for each instructor at the school.	InstructorID* Name RoomNo

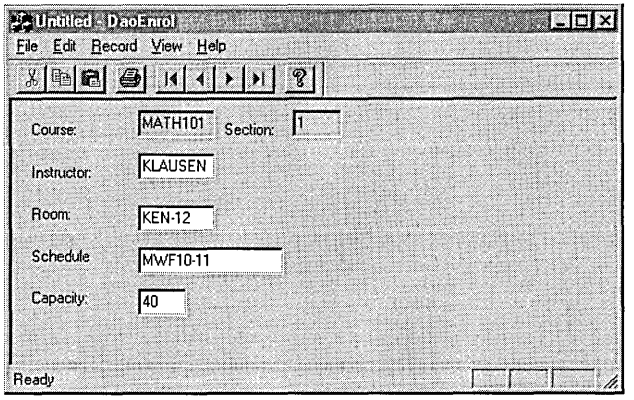
*Indicates the column (or columns) that comprise the table's primary key.

†The Dynabind_Section table is used in the Dynabind sample, but not in the DaoEnrol tutorial.

Note You can use Books Online to copy STDREG32.MDB to your local drive. You can also easily install the sample source project files for DaoEnrol. For more information, see "Installing the Sample Files."

DaoEnrol lets you use a "form" — a view with dialog-style controls — to view registration information for courses, section by section. Section information displayed includes the course name, section number, instructor, room, and schedule (such as "MWF 10-11"). For example, you can view section 1 of the course MATH 101, then section 2, and so on. The initial tutorial step provides read-only viewing of all sections. Steps 2 and 3 add more capabilities, including updates. Figure 34.1 shows what the DaoEnrol application looks like at the end of the tutorial.

Figure 34.1 The Completed DaoEnrol Tutorial Application



Setting Up the Student Registration Data Source for DaoEnrol

Before you begin the DaoEnrol tutorial you must specify a database. You can either:

- Copy and use the prebuilt STDREG32.MDB database

  –or–

- Create your own database

MFC support for Data Access Objects (DAO) does not rely on Open Database Connectivity (ODBC) for most database formats. Because of the flexibility of the DAO database classes, you have several options for creating and using the DaoEnrol database:

- The Microsoft Jet database engine works natively with Microsoft Access .MDB files, but it can also directly read ISAM databases such as Paradox, dBase, and FoxPro. For better performance, you will want to attach these external data sources as tables in a Microsoft Access .MDB file. For more information, see "DAO External: Attaching External Tables."

  While you can create a database, tables, queries, fields, indexes, and recordsets with DAO, it is much easier to just use the STDREG32.MDB database supplied with Visual C++ for this tutorial. You can use Books Online to copy STDREG32.MDB to your local drive.

  Note When you use AppWizard to create a DAO database application, the only database type offered as a selection is a Microsoft Access .MDB file. This tutorial assumes that you will use the STDREG32.MDB file. If you use another database type, you must use Microsoft Access to create an .MDB file, attach the database table to that .MDB file, and select that .MDB file during the selection of the Data Source in AppWizard–Step 2.

- If you want to use another database format, you must install the corresponding 32-bit ODBC driver, as well as its related database management system (DBMS). (Microsoft Visual C++ ships 32-bit ODBC drivers for most standard database formats; for a complete list, see "ODBC Driver List" in *Programming with MFC*). Use the DBMS to add tables to your database so that it matches the Student Registration database schema. Then, use ODBC Administrator to register the database using the instructions in the Enroll tutorial in Chapter 30. For more information about using ODBC Administrator, see the articles "ODBC Administrator" and "Data Source (ODBC)," in *Programming with MFC.*

  Note When you run a Typical Visual C++ Setup, the ODBC drivers for dBase, FoxPro, Access and SQL Server are installed automatically. You can rerun Setup (or choose a Custom installation the first time you run Setup) to install drivers for Paradox, Microsoft Test, Excel, and Oracle. If you need an ODBC driver not provided with Visual C++, you must use the installation program shipped with the program to install it, and use ODBC Administrator to register it.

Specify a Database for DaoEnrol

The easiest way to supply a database for the DaoEnrol tutorial is to use the pre-built STDREG32.MDB Microsoft Access database file, included with Visual C++ for this purpose. Alternatively, you can create your own database.

▶ To use STDREG32.MDB with DaoEnrol

1. From InfoView, expand the following folders:

   Samples \ MFC Samples \ Databases (ODBC and DAO)

2. Double-click the page node for the STDREG sample.
3. In the STDREG topic, click the button provided to copy sample project files.
4. In the Sample Application dialog box, select STDREG32.MDB and choose Copy.
5. In the Copy dialog box, navigate to the directory where you want to copy this file, and click OK.

   Visual C++ creates the directory for you, if necessary, and copies the file. STDREG32.MDB already contains the tables and records used in the tutorial.

▶ To create your own database for DaoEnrol

- Create a new database schema using the database administration capability of your DBMS.

  Depending on the type of DBMS, you might create the new database on a server that is different from the machine where you will be doing MFC database development. In either case, you need to add tables to the new database to match the schema of STDREG32.MDB. You must also install the ODBC driver that corresponds to your DBMS, and register the database with the ODBC Administrator.

DAO Tutorial Steps

The following table briefly describes the DaoEnrol tutorial steps:

Tutorial Step	Chapter	Description
1	34 (this chapter)	Use AppWizard to create an application with database support. The document embeds a **CDaoRecordset** object for the Section table of the Student Registration data source. Use the dialog editor to design the form. Use ClassWizard to bind controls on the form to fields in the recordset.
2	32	Provide a combo box control on the form so the user can select a course and view its sections. Fill the combo box from a recordset object representing the Course table. Filter and parameterize the recordset to constrain the records it selects. (Use Enroll Step 2 with substituted sections from DaoEnrol Step 2.)
3	33	Implement a user interface for adding, updating, and deleting records. Handle database exceptions. (Use Enroll Step 3 with substituted sections from DaoEnrol Step 3.)

DaoEnrol Step 4: A Preview

The sample source code for DAOENROL consists of the code for Steps 1, 2, and 3, plus a Step 4 not covered in the tutorial. Step 4 illustrates additional database programming techniques as summarized below. See DAOENROL under Samples in Books Online for a discussion of DaoEnrol Step 4. The techniques illustrated by DaoEnrol Step 4 include:

- Using multiple record view classes.
- Switching views in a frame window.
- Using the document object to coordinate multiple forms via **UpdateAllViews** and update hints.

A Brief Overview of DAO

MFC version 4.0 includes new database classes for programming with Data Access Objects (DAO). DAO is an application programming interface (API) based on OLE.

In general, the DAO database classes offer more complete database functionality than the ODBC database classes, which were first introduced in MFC 2.5. For a more detailed overview of DAO, see the articles "Database Overview," "DAO and MFC," and "Data Access Objects (DAO)," in *Programming with MFC.*

DAO supplies a hierarchical set of objects that use the Microsoft Jet database engine to access data and database structure in:

- Microsoft Jet (.MDB) databases.
- ODBC data sources, using an ODBC driver.
- Installable ISAM databases, such as dBASE, Paradox, FoxPro, and Btrieve, which the database engine can read directly.

DAO can access other types of databases through ODBC. For more information about using the DAO database classes versus using the ODBC database classes, see Chapter 7, "Working with Databases" in *Programming with MFC*.

Not all DAO objects are exposed in MFC, although most of the DAO functionality is available. You can make direct calls to DAO, and the DAO Software Development Kit (SDK) and DAO Help are included in Visual C++ version 4.0. For more information on the relationship between DAO and MFC, see the articles "DAO Collections," "DAO and MFC," "DAO: Writing a Database Application," and "DAO: Database Tasks" in *Programming with MFC*.

Data Access Objects in MFC

The DAO objects encapsulated by MFC are listed below in hierarchical order.

Workspace

In DAO, the Workspace object defines a session for a user. It contains open databases and provides mechanisms for simultaneous transactions. MFC includes access to the DBEngine object that controls the Microsoft Jet database engine and manipulates its properties. For more information on MFC's Workspace object, see class **CDaoWorkspace** in the *Class Library Reference*, and the articles "DAO Workspace," and "DAO Workspace: The Database Engine" in *Programming with MFC*.

Database

The Database object represents a connection to the database. The DAO database classes have greater ability to manipulate databases than do the ODBC database classes. DAO can read Microsoft Access .MDB files directly. DAO can also read installable ISAM databases directly (dBASE, Paradox, FoxPro, and Btrieve) if they are used as attached tables. Other databases, such as Oracle and SQL Server, can be read by using ODBC and their corresponding drivers. For more information, see class **CDaoDatabase** in the *Class Library Reference*, and the articles "DAO Database," and "DAO External: Working with External Data Sources" in *Programming with MFC*.

Tabledef

The Tabledef object represents the schema (structure) of a table. The DAO database classes let you create and manipulate tables in a database using the Data Manipulation Language (DML) subset of SQL. You can use the DAO database classes to create and delete fields (columns) in a table and to create and delete indexes for a table. For more information, see class **CDaoTableDef** in the *Class Library Reference*, and the article "DAO Tabledef" in *Programming with MFC*.

Querydef

Most of the applications you create require only a subset of database records. The Querydef object represents a query you create and execute, or use to create a recordset, filtering and sorting records as desired. You can also create and execute action queries and SQL pass-through queries. For more information, see class **CDaoQueryDef** in the *Class Library Reference*, and the articles "DAO Querydef" and "DAO Querydef: Action Queries and SQL Pass-Through Queries" in *Programming with MFC*.

Recordset

A Recordset object represents a set of records selected from a data source. The recordset may represent a selection of one or more specified columns from rows of one or more database tables. A **CDaoRecordset** object represents both (a) this selection of records and (b) the actual field values for one currently selected record.

For more information, see class **CDaoRecordset** in the *Class Library Reference*, and the articles "DAO Recordset," "DAO Recordset: Recordset Navigation," "DAO Record Field Exchange (DFX)," and "DAO Record Field Exchange: Double Buffering Records" in *Programming with MFC*.

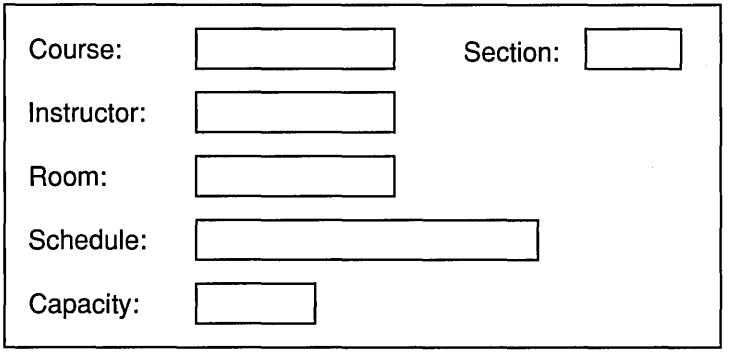
Exceptions

DAO error handling is accomplished through the use of MFC exceptions. For more information, see class **CDaoException** in the *Class Library Reference*, and the article "Exceptions: Database Exceptions" in *Programming with MFC*.

DaoEnrol Step 1

Step 1 of the DaoEnrol tutorial implements an updatable database form that lets the user examine the records in the Section table one record at a time. You'll create a form that looks like the one shown in Figure 34.2 .

Figure 34.2 DaoEnrol's Section Form



In Step 1 you will:

- Create the DaoEnrol application.
- Examine the DaoEnrol Step 1 classes.
- Customize DaoEnrol's database form.
- Bind DaoEnrol's form controls to recordset fields.
- Build and run DaoEnrol Step 1.

If you choose to work along with the tutorial, perform all the steps in the procedures in this chapter. At the end, you'll be able to build and run the DaoEnrol Step 1 application.

Creating a New DAO Database Application

When you use AppWizard to create a DAO database application, the only database type offered as a selection is the Microsoft Access .MDB database. Using either Access data in an .MDB file or an installable ISAM database as an attached table to an .MDB file gives your application the best performance for data retrieval and manipulation. The procedures in this tutorial assume that you are using the STDREG32.MDB file supplied on the Visual C++ distribution CD.

For more information on how to use the AppWizard Database Options page in AppWizard when you're creating your starter application, see "AppWizard: Database Support" in *Programming with MFC*. AppWizard lets you specify whether your database application uses a file as well as a database. The DaoEnrol application does not need a file, so it is based on the "Database view without file support" option in AppWizard.

For more information about applications that do not use file support, see "Serialization: Serialization vs. Database Input/Output" in *Programming with MFC*.

Note The following procedure describes how to enter the correct values in the AppWizard Steps to create the DaoEnrol application. Many of the Steps contain choices that you will not use to create the starter files for DaoEnrol. For more information on the various options that appear in these Steps, see Chapter 1, "Creating Applications Using AppWizard," in the *Visual C++ User's Guide*.

▶ To create the tutorial database application

1 From the File menu, choose New.

The New dialog box appears.

2 Select Project Workspace.

The New Workspace dialog box appears.

3 In the Project Name box, type `DaoEnrol`.

AppWizard creates a project directory with this name under the main (root) directory specified in the Location box.

4 In the Type list box, make sure MFC AppWizard (exe) is specified.

5 If necessary, use the Location box to specify a different root directory for the DaoEnrol project files.

Depending on the directory you last worked in, you may want to change where the Location box currently points. You can use the Browse button to navigate to an existing directory, or type a directory name directly into the Location box. AppWizard creates this directory if it doesn't exist.

6 If any check boxes other than Win32 appear in the Platforms box, clear them.

7 Click Create.

AppWizard creates the project directory, and MFC's AppWizard–Step 1 appears. Click the Single Document Interface radio button.

8 Click Next to continue to Step 2.

This is the AppWizard database options Step.

9 In AppWizard–Step 2 (the database options page):

- Select the "Database view without file support" option.

  This enables the Data Source button.

- Click the Data Source button.

  The Database Options dialog box appears.

- Click the radio button to select the DAO option. click the browse button (...) to open the Open dialog. Navigate to and select the Microsoft Access file STDREG32.MDB, then click OK.

- AppWizard uses Dynaset as the default recordset type. The check boxes for Dirty Fields and Bind All Columns are also checked by default. click OK. This returns you to AppWizard–Step 2. The Select Database Tables dialog box appears.
- Select the table name Section, and click OK.

10 Click Next to proceed with the rest of the AppWizard Steps, and click Next in AppWizard for Steps 3, 4 and 5 to accept the default options.

In the AppWizard–Step 6 dialog box, you can check and, if necessary, modify the default names that AppWizard creates for your program's classes and files.

Note By default, AppWizard bases the names of classes on the project name you supply. This naming is probably fine if your application has only one recordset/record view pair. If your application has multiple recordsets and record views, it's a good idea to change the name of the first recordset/record view pair created by AppWizard so the naming better reflects the name of the table in the data source. For DaoEnrol, you'll modify two class names and their related header and implementation file names, even though the tutorial uses only one recordset/record view pair.

11 In AppWizard–Step 6, make the following changes to class names:

- Select the class `CDaoEnrolDoc`, and change the header file name to DENRLDOC.H. Change the implementation file name to DENRLDOC.CPP.
- Select the class `CDaoEnrolView`, and change its name to `CSectionForm`. Change the header filename to SECTFORM.H. Change the implementation file name to SECTFORM.CPP.

  The base class is **CDaoRecordView**.
- Select the class `CDaoEnrolSet`, and change its name to `CSectionSet`. Change the header filename to SECTSET.H. Change the implementation file name to SECTSET.CPP.

  The base class is **CDaoRecordset**. The edit item is disabled to show that you can't change it.

12 Choose Finish.

The New Project Information dialog box appears, summarizing the settings and features AppWizard will generate for you when it creates your project.

You might want to take a moment to examine the application type, classes, and features that AppWizard automatically provides.

13 Click OK in the New Project Information dialog box.

AppWizard creates all necessary files, and opens the project.

Use ClassView to view the classes that AppWizard just created. The next section, "Examining the DAOEnrol Step 1 Classes," describes this in more detail.

Examining the DaoEnrol Step 1 Classes

Once AppWizard creates the DaoEnrol project, you can use ClassView to see a graphical representation of the classes and any default member functions AppWizard created. You can also use ClassWizard to view member variable bindings that AppWizard specifies for you.

The CSectionSet Recordset Class (DAO)

The following procedure describes how to view the new recordset class, `CSectionSet`. After examining `CSectionSet`, you'll use the text editor to examine the source files for classes `CSectionForm` and `CDaoEnrolDoc`.

▶ To examine the new recordset class

1. In ClassView, expand the DaoEnrol folder.

   Notice the rich set of classes AppWizard created for you automatically to support the DaoEnrol application.

2. From the list of classes, choose `CSectionSet`.

   ClassView displays all the member variables that AppWizard created, including a variable for each of the Section table's columns. You can use ClassWizard to view how AppWizard has bound the Section table's columns to these member variables.

3. From the View menu, choose ClassWizard.

4. Choose the Member Variables tab.

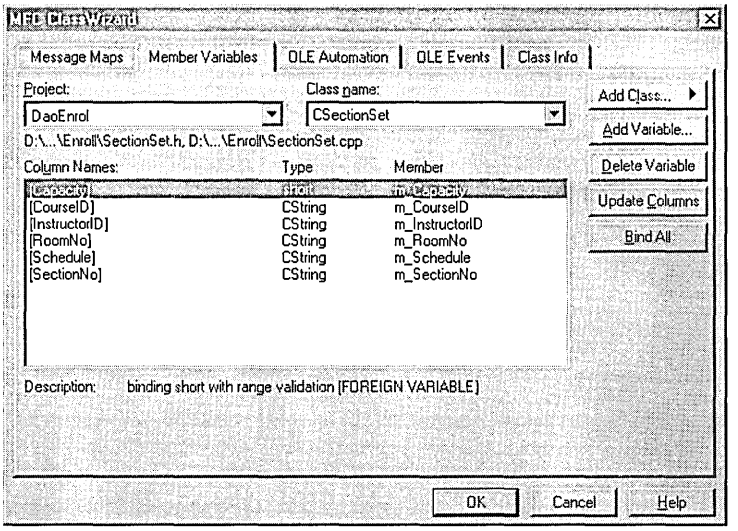
5. In the Class Name box, select CSectionSet.

   What you see in this tab corresponds to the variables displayed in ClassView. AppWizard has bound all of the table's columns to member variables of the `CSectionSet` class. These member variables are called "field data members." AppWizard names the data members automatically, based on the column names from the data source. AppWizard also assigns the correct C++ or class library data type to the data members, based on the column type. In this example, all of the columns are text columns, mapped to type **CString**, except the Capacity column, which is a **short**.

6. Once you've finished examining the `CSectionSet` recordset class, click Cancel to exit ClassWizard.

Figure 34.3 shows the ClassWizard Column Names list.

Figure 34.3 Table Columns Mapped to Recordset Data Members



If you don't want all of a table's columns bound to your recordset, you can delete the recordset field data members for those columns you don't want by selecting the data member and clicking the Delete Variable button. For the tutorial, you will need them all.

Caution Don't delete any fields that are part of the table's primary key (in this case, the SectionNo and CourseID fields).

To change the name of a field data member, delete the member and add it again with the new name. For more information, see "ClassWizard: Binding Recordset Fields to Table Columns" in *Programming with MFC*.

In the next procedure, you'll examine the `CSectionForm` record view class from inside the text editor.

The CSectionForm Record View Class (DAO)

As stated earlier, `CSectionForm` is derived from **CDaoRecordView**, which is one of the MFC record view classes. A record view is a specialized class that uses controls laid out in a dialog template resource to view and/or edit the fields of a recordset in a dialog-like form. A **CDaoRecordView** object is associated with both (a) a recordset object and (b) a dialog template resource. The dialog template resource has an ID of the form **IDD_XXX_FORM**, where XXX is based on the project name.

CDaoRecordView derives its behavior from class **CFormView**. **CDaoRecordView** supports navigation through records, one record at a time, using commands in the associated **CDaoRecordset** object. When you update the value in a control on the form and navigate to another record, the corresponding recordset field is automatically updated. For more information, see the article "Record Views" in *Programming with MFC*.

▶ To examine the source code for the record view class

- Use ClassView to jump to the `OnInitialUpdate` member function of class `CSectionForm`.

  SECTFORM.CPP opens inside the text editor, with your cursor just before the `OnInitialUpdate` member function:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;
    CDaoRecordView::OnInitialUpdate();
}
```

  The base class framework function **CDaoRecordView::OnInitialUpdate** opens the database if it is not already open, then opens the recordset, and initializes the form by calling **CFormView::OnInitialUpdate**.

For now, the "form" represented by class `CSectionForm` is empty of controls. Later, in "Customizing the Dialog Template for the DaoEnrol Section Form," you'll use the dialog editor to design the form and to map controls on the form to the recordset.

The CDaoEnrolDoc Document Class

In ClassView, you can see that AppWizard created a class, `CDaoEnrolDoc`, derived from **CDocument**.

▶ To view the document class

- In ClassView, double-click the icon for class `CDaoEnrolDoc`.

  This opens the file DENRLDOC.H in the text editor, placing the insertion point beside the `CDaoEnrolDoc` class declaration.

What is the document's role in a database application? In most other applications, the document stores data and serializes it to a file on disk. Often the application reads the whole file into memory at once and writes it back to disk as a whole. In a database application, however, the data is stored in the database, and the user usually views the data as records. Such an application doesn't need a file.

A document in a database application, then, isn't normally used for its serialization support. So why does DaoEnrol have a document class?

The following code, at the beginning of DENRLDOC.H, reveals that the role of the document class in DaoEnrol is to own the recordset.

```
class CDaoEnrolDoc : public CDocument
{
    ...
    // Attributes
    public:
        CSectionSet m_sectionSet;
    ...
};
```

The recordset object, `m_sectionSet`, is embedded in the document object. Therefore, the recordset object is automatically constructed when the document object is constructed, and automatically deleted when the document object is deleted.

The document class can own any number of recordset objects in this way. For example, Step 4 of DaoEnrol adds a second form and corresponding recordset; the document embeds this second recordset.

In a sense, then, the document class is a proxy for the database. This approach isn't strictly necessary, but if you (or AppWizard) design your database application to use the document class this way, you can better take advantage of the framework's document/view architecture. For example, if you have multiple views (forms) simultaneously showing some of the contents of the database, you can take advantage of the **CDocument::UpdateAllViews** mechanism to conveniently notify all views about an update that might have been initiated in one of the views.

If you look at the menu resource that AppWizard created when you chose the option "Database view without file support," you'll see that there are no New, Open, Save, or Save As commands on the File menu. The File menu has only the Print, Print Preview, Print Setup, and Exit commands. If you had chosen "Database view with file support," AppWizard would have supplied the missing File menu commands.

▶ To view the DaoEnrol menu resource

1. From the ResourceView, expand the DaoEnrol folder.

   This displays the resource browser, which shows the resources associated with a project.

2. In the resource browser, expand the Menu folder.

3. Double-click IDR_MAINFRAME.

   The Menu Editor opens, displaying the default menu that AppWizard created for the DaoEnrol application.

4. Click the File menu item to view its structure. Notice the absence of New, Open, Save, and Save As.

5. Close the Menu Editor when you're finished.

Note If you choose the option "Database view with file support" in AppWizard, the document class plays two roles. First, it serves as a proxy for the database. Second, it represents the file that is opened and saved via the New, Open, Save, and Save As commands on the File menu. This file might be used for a variety of purposes; for ideas, see "MFC: Using Database Classes With Documents and Views" and "Serialization: Serialization vs. Database Input/Output" in *Programming with MFC.*

For more information about documents and views, see Chapter 3, "Working with Frame Windows, Documents and Views," and Chapter 4, "Working with Dialog Boxes, Controls, and Control Bars" in *Programming with MFC.*

Customizing the Dialog Template for the DaoEnrol Section Form

Along with the classes, AppWizard creates a dialog template resource named `IDD_DAOENROL_FORM`, which the **CDaoRecordView**-derived class, `CSectionForm`, uses to display its form controls. Because **CDaoRecordView** is derived from **CFormView**, a record view's client area is laid out by a dialog template resource. The layout of the form is up to you. AppWizard places one static text control on the dialog template resource, labeled "TODO: Place form controls on this dialog."

In the following procedure, you'll replace this text with controls that correspond to columns in the table (via the field data members of the recordset).

▶ To customize DaoEnrol's form

1. In the ResourceView, double-click the DaoEnrol folder.
2. Expand the Dialog folder.
3. Double-click IDD_DAOENROL_FORM.

   The dialog editor opens and displays the dialog box with the corresponding ID.

   For more information about the dialog editor, see Chapter 6, "Using the Dialog Editor," in the *Visual C++ User's Guide.*
4. Select, and then delete the static control that says "TODO: Place form controls on this dialog."
5. Design DaoEnrol's Section form to resemble Figure 34.4, using static controls and edit controls.

   Tip Press ALT+ENTER to display the Properties window, and then pin it down so it stays open while you add and edit the dialog box controls.

   Resize the dialog box as needed. You may want to add the controls in pairs, for example: static text control, then the corresponding edit control, and so on.

6 For each edit control, use the ID box in the Properties window to specify an ID based on the table column names (for example, `IDC_COURSE`). This is only a convention, but it is used throughout the tutorial.

Note The "Edit" caption that appears in each edit control is not visible to the user at run time, so you needn't worry about deleting it. To view the dialog box as it will appear at run time, press CTRL+T to enter test mode for the dialog box. Press ESC to cancel test mode.

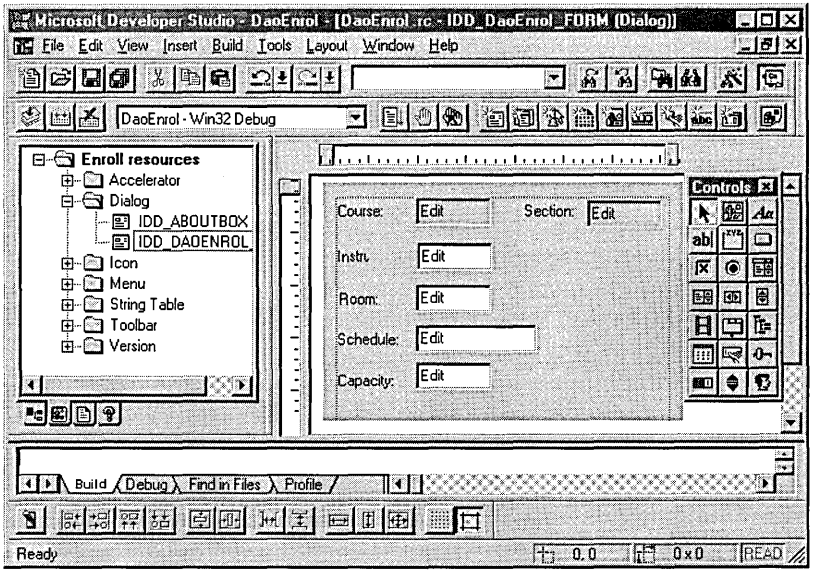
7 Make the Course and Section edit controls read-only. To do so, select the Styles page in the Properties window and set the Read-Only check box. (The other edit controls are updatable.)

According to a common rule in the user interface design of database forms, the user shouldn't be able to update these key fields. If users want to change a course number or section of a Section record, they must delete the old Section record and add a new one to avoid possibly violating the referential integrity of the database. DaoEnrol Step 3 implements Add and Delete functionality.

8 Save the DaoEnrol Resources (DAOENROL.RC).

It's a good idea to periodically back up your work.

Figure 34.4 The Layout of DaoEnrol's Section Form



If you did not add the static text and corresponding edit controls in order, one after the other, you need to change the tab order. You can easily check the current tab order and change it if necessary.

▶ To view or change the tab order of controls

1 With the dialog resource open, from the Layout menu choose Tab Order.

You'll see numbers depicting the current tab order of the controls.

2 Specify the tab order you want by clicking each control in that order.

As you click, you'll see the numbering change to reflect your choices.

For DaoEnol, specify a tab order such that each edit control is preceded in the tab order by the static text control that describes it. By specifying this tab order, you enable ClassWizard to derive a name for the edit control when you bind it to a data member, as you'll do in the next section.

Binding DaoEnrol's Controls to Recordset Fields

With the form designed, it's time to indicate which edit controls map to which table columns—or, more precisely, which controls map to which recordset field data members. To perform this task, you use ClassWizard's "foreign object" mechanism. (For details about foreign objects, see the article "ClassWizard: Foreign Objects" in *Programming with MFC.*)

Normally, you use ClassWizard to bind controls in a dialog box or form to member variables of your **CDialog**- or **CFormView**-derived class. In the case of **CDaoRecordView**, though, you bind the form's controls not to data members of the record view class but to data members of the recordset class associated with the record view.

Your **CDaoRecordView**-derived class—`CSectionForm` in this case—has a data member called `m_pSet`. You can view `m_pSet` in the ClassView of `CSectionForm`. This data member is a pointer to `CSectionSet`, DaoEnrol's recordset class. Recall that you viewed this recordset class in the Member Variables tab of ClassWizard (see Figure 34.3).

The control bindings go through `m_pSet` to the corresponding field data members of `CSectionSet`. For example, in the following procedure, you will bind the Course edit control to:

```
m_pSet->m_CourseID
```

▶ To bind a form control to a recordset data member

1 If necessary, choose IDD_DAOENROL_FORM from the resource browser to open the dialog box inside the dialog editor.

For more information about the dialog editor, see Chapter 6 in the *Visual C++ User's Guide.*

2 In the dialog editor window, hold down the CTRL key and double-click the Course edit control.

ClassWizard's Add Member Variable dialog box appears, with a proposed field name selected for you in the Member Variable Name box. ClassWizard chooses this name based on the caption of a static text control that falls previous to the edit control in the tab order.

For example, for `IDC_COURSE`, the control's caption is "Course," and the Member Variable Name box should display:

```
m_pSet->m_CourseID
```

3 Click OK in the Add Member Variable dialog box to accept the name.

4 Repeat steps 2 and 3 for each of the other edit controls on the form.

It isn't necessary to create mappings for the static text controls.

5 Save your work.

Note Using CTRL+double-click in the dialog editor is a ClassWizard shortcut for mapping form controls to members of the associated dialog box, form view, or record view class. Use it on a pushbutton to create a command handler function for the button. Use it on other controls to create a class member variable.

You can view the complete mappings for class `CSectionForm` in the ClassWizard Member Variables tab. For example, where `IDC_COURSE` appears in the Control IDs column, you'll see `->m_CourseID` in the corresponding Member column.

Build and Run DaoEnrol Step 1

Build and run DaoEnrol Step 1. For information on building, see "Build the Starter Application," in Chapter 3.

When the `CSectionSet` recordset opens, it selects records from the Section table in the Student Registration database. The first record becomes the "current record" in the recordset. DaoEnrol's database form displays the controls you designed, now filled with data from the current record.

Here are some things to try:

- Take a look at the Record menu, which has First Record, Previous Record, Next Record, and Last Record commands. (The toolbar has buttons that correspond to the menu commands.) Try using the commands to scroll through the records in the recordset.
- Try updating some of the fields. The new values are accepted into the database when you move to another record. As mentioned earlier, the key fields Course and Section are read-only.

Important You must have write access to the database in order to update records. You can use File Manager to view the database properties and, if necessary, clear the Read-Only checkbox.

When you finish, exit the program.

This completes Step 1 of the DaoEnrol. To continue with this tutorial, use the instructions in Chapters 32 and 33 to complete the application. See "DaoEnrol Step 2" and "DaoEnrol Step 3" for more information.

Completing the DaoEnrol Tutorial

Most of the procedures you follow to create the Enroll and DaoEnrol applications are identical. Some of the implementation details of DAO, however, require substitutions for some of the procedures you use in Chapters 32 and 33 (Steps 2 and 3 of Enroll). The procedures you use instead are provided in "DaoEnrol Step 2" and "DaoEnrol Step 3." DaoEnrol Step 4 is the DAOENROL sample.

DaoEnrol Step 2

Chapter 32, "Using a Second Recordset," continues both the DaoEnrol and Enroll tutorials by showing you how to add a second recordset and use it to fill a combo box control on the form. Follow the sections in Chapter 32 and the substitutions listed below to complete Step 2 of DaoEnrol.

In Chapter 32:

- "Using a Second Recordset"
- "About Step 2"
- "Changing the Course Control to a Combo Box"
- "Binding the Combo Box Control to a Recordset Field and a CComboBox Variable"

Next, in Chapter 34:

- "Creating a Recordset for the Course Table in DaoEnrol"

  The steps for selecting a data source are different for a DAO-based database application.
- "Embedding the Recordset Object in the Document Object in DaoEnrol"

  The names of files you use differ in DaoEnrol.
- "Filling the Combo Boxes in DaoEnrol"

  A substitute code block for DaoEnrol is provided because the DAO database classes handle parameterizing a query differently than the ODBC database classes.

- "Filtering and Parameterizing the Recordset in DaoEnrol"

  This section does not add code to the tutorial, but provides details on how the DAO database classes handle parameterized queries.

- "Parameterizing the Filter in DaoEnrol"

  Discussion of how the DAO database classes handle parameters.

Finally, finish in Chapter 32:

- "Reusing a Database Object Opened by Another Recordset"
- "Sorting the Recordset"
- "Requerying the CSectionSet Recordset"

Note The figures in Chapter 32 show "Enroll" in the title bar of the mainframe window. Your application will show "DaoEnrol". Remember to substitute **CDaoRecordset** for **CRecordset**, **CDaoRecordView** for **CRecordView**, and so on.

Creating a Recordset for the Course Table in DaoEnrol

Note Before starting this section, you should have already completed the instructions in Chapter 32 for "Using a Second Recordset" through "Binding the Combo Box Control to a Recordset Field and a CComboBox Variable."

DaoEnrol already has one recordset, for the Section table, which fills the controls on the `CSectionForm` record view with information about a single class section of the currently selected course name. Now you'll add a second recordset, for the Course table, used to fill the combo box control with a list of all available course names.

▶ To create a new recordset class

1. From the View menu, choose ClassWizard.
2. Click the Add Class menu button, and from the menu select New.

   This opens the Create New Class dialog.
3. In the Name box under Class Information, type `CCourseSet`.
4. From the Base Class drop-list, select CDaoRecordset.
5. Click the Change push button in the File group box, shorten the filenames to COURSESE.H and COURSESE.CPP, and click OK.
6. Clear the Add to Component Gallery checkbox.

   For more information about this option, see "Using Component Gallery" in the *Visual C++ User's Guide*.
7. Click Create.

   This opens the Database Options dialog box.

▶ To connect the recordset class to the Course table

1 Click the browse button [...] next to the DAO DataSource option.

This displays the Open dialog box.

2 Navigate to the STDREG32.MDB file and select it. Click OK to return to the Database Options dialog box, then click OK again.

The Select DatabaseTables dialog box opens.

3 Select the table name "Course," and click OK.

This connects the table name to class `CCourseSet` and returns you to the ClassWizard Member Variables tab. The Class Name box shows `CCourseSet`, and three names are listed in the Column Names box. Table 34.2 shows the column names, their data members, and their data types.

4 Click OK to close ClassWizard.

Note On the Member Variables tab, you can see that all of the table's columns are already assigned to field member variables. You can use ClassWizard to delete those variables if you don't need to access or modify the columns—but be careful not to delete a field member variable for a column that is part of the table's primary key.

Table 34.2 CCourseSet Data Members

Column name	Type	Data member
CourseID	**CString**	`m_CourseID`
CourseTitle	**CString**	`m_CourseTitle`
Hours	**short**	`m_Hours`

For more information about using ClassWizard to create recordset classes, see the article "ClassWizard: Creating a Recordset Class" in *Programming with MFC*.

Embedding the Recordset Object in the Document Object in DaoEnrol

In DaoEnrol Step 1, AppWizard embedded the `CSectionSet` object in the document. In this step, you'll do the same for the second recordset object—an object of the `CCourseSet` class that you created earlier with ClassWizard.

▶ To embed the recordset in the document

1 From FileView, open the Dependencies folder and double-click on the file DENRLDOC.H to open it.

2 Declare an embedded `CCourseSet` object by adding the following line to the public Attributes section, just beneath the `CSectionSet` declaration:

```
CCourseSet  m_courseSet;
```

3 Similarly, open files DENRLDOC.CPP, DAOENROL.CPP, and SECTFORM.CPP, and add a **#include** directive for "COURSESE.H" before the existing **#include** directive for "DENRLDOC.H", as shown in the following line:

```
#include "CourseSe.h"
```

4 Save DENRLDOC.H and the .CPP files.

The document's `m_courseSet` member is referred to in the implementation of `OnInitialUpdate` that you'll complete later.

Filling the Combo Box in DaoEnrol

A good place to fill the combo box with a list of course names is the `CSectionForm` override of **CDaoRecordView**'s **OnInitialUpdate** member function. As part of its own initialization, the form fills the combo box. The overall logic is as follows:

1. Construct and open a `CCourseSet` recordset based on the Course table.
2. Remove any current entries in the combo box.
3. For each course name in `CCourseSet`, add the CourseID to the combo box.
4. Set the selection to the first course name (as sorted) in the combo box.

The code in the following procedure fills the combo box and also filters, parameterizes, and sorts the `CSectionSet` recordset. Filtering, parameterization, and sorting are explained in sections that follow.

▶ To fill the combo box

1 From the Window menu, select the file SECTFORM.CPP (this file should still be open from the previous procedure), or use FileView to find and open the file.

2 Use ClassView or WizardBar to navigate to the implementation of `OnInitialUpdate`.

3 Just after the first line — `m_pSet = &GetDocument()->m_sectionSet;` — add the code below (don't replace any code):

```
// Fill the combo box with all of the courses
CDaoEnrolDoc* pDoc = GetDocument();
pDoc->m_courseSet.m_strSort = "CourseID";

try
{
    pDoc->m_courseSet.Open();
}
catch(CDaoException* e)
{
    AfxMessageBox(e->
        m_pErrorInfo->m_strDescription);
    e->Delete();
    return;
}
```

```
// Filter, parameterize and sort the
// CSectionSet recordset

m_pSet->m_strFilter =
    "CourseID = CourseIDParam";
m_pSet->m_strCourseIDParam =
    pDoc->m_courseSet.m_CourseID;
m_pSet->m_strSort = "SectionNo";
m_pSet->m_pDatabase =
    pDoc->m_courseSet.m_pDatabase;
```

You'll add the necessary member declaration in the section "Parameterizing the Filter in DaoEnrol."

4 Next, after the last line (`CDaoRecordView::OnInitialUpdate();`), add the following code:

```
m_ctlCourseList.ResetContent();
if (pDoc->m_courseSet.IsOpen())
{
    while (!pDoc->m_courseSet.IsEOF())
    {
        m_ctlCourseList.AddString(
            pDoc->m_courseSet.m_CourseID);
        pDoc->m_courseSet.MoveNext();
    }
}
m_ctlCourseList.SetCurSel(0);
```

5 Save your work.

For more information, see the article "Record Views: Filling a List Box from a Second Recordset" in *Programming with MFC*.

Filtering and Parameterizing the Recordset in DaoEnrol

The Step 1 version of DaoEnrol selects all of the records in the Section table into `CSectionSet`. In Step 2, only the class sections for a specific course name should be selected. This discussion introduces the concepts of recordset filters and parameters.

Setting Up the Filter in DaoEnrol

Note You've already added the code to filter and parameterize the `CSectionSet` recordset (in `OnInitialUpdate`); the code in this section is for illustrative purposes only. Do not add the code from this section to your source files.

A recordset filter determines what subset of records are selected from a table or query. To add a filter, you simply set the value of **CDaoRecordset::m_strFilter** before calling **CDaoRecordset::Open**. For example, the following code selects just the class section records for course MATH101:

```
m_pSet->m_strFilter = "CourseID = 'MATH101'";
m_pSet->Open();
```

Since the base class **CDaoRecordView::OnInitialUpdate** calls **CDaoRecordset::Open**, all you need to do to initially select the records for MATH101, for example, is replace the following AppWizard implementation of `OnInitialUpdate`:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;
    CDaoRecordView::OnInitialUpdate();
}
```

with:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;
    m_pSet->m_strFilter = "CourseID = 'MATH101'";
    CDaoRecordView::OnInitialUpdate();
}
```

The filter can be any logical expression that is legal for the SQL **WHERE** clause. For example, the following is legal:

```
m_pSet->m_strFilter =
                "CourseID = 'MATH101' AND InstructorID = 'ROGERSN'";
```

As an alternative, you can specify a complete SQL **SELECT** statement with a **WHERE** clause. In that case, you don't use **m_strFilter**. For more information about filtering recordsets in the MFC DAO classes, see the article "DAO Queries: Filtering and Parameterizing Queries" in *Programming with MFC*.

Examine the `OnInitialUpdate` code you added earlier. It shows the filter for `CSectionSet` in Enroll Step 2.

Parameterizing the Filter in DaoEnrol

DaoEnrol reselects, or "requeries," class section records every time the user selects a new course name from the combo box. One way to implement this is to close the old `CSectionSet` object and reopen it, supplying a new **m_strFilter** value before calling **Open**. This works, but is somewhat inefficient, because the framework has to completely reconstruct and run a new SQL **SELECT** statement. A more efficient way to requery the same recordset is to "parameterize" the filter—call **Requery** with a new filter value and a specific parameter value.

In order to parameterize the filter, you'll perform the following procedures:

- Declare a parameter data member in the recordset's header file.
- Bind the parameter data member to the recordset.

To implement the **Requery** with a new filter and a specific parameter value supplied at run time, you:

- Specify a parameterized filter.
- Supply the run-time parameter value.

The following sections describe these procedures.

▶ To declare a parameter data member in the recordset's header file

1 Open the SECTSET.H file.

2 Add the following member variable declaration for `m_strCourseIDParam`, just before the `//Overrides` section, after the `//}}AFX_FIELD` line:

```
CString m_strCourseIDParam;
```

▶ To bind the parameter data member to the recordset

1 Use ClassView to navigate to the `CSectionSet` constructor, and initialize the parameter count variable, **m_nParams**, which by default is zero. Also initialize DaoEnrol's single parameter, `m_strCourseIDParam`.

Place the following two lines of code after the line `m_bCheckCacheForDirtyFields = FALSE;:`

```
m_nParams = 1;
m_strCourseIDParam = "";
```

2 Use ClassView or WizardBar to navigate to the `DoFieldExchange` member function definition, and add the following two lines of code to identify `m_strCourseIDParam` as a parameter data member. Add the code at the end of the function, after the `//}}AFX_FIELD_MAP` line.

```
pFX->SetFieldType(CDaoFieldExchange::param);
DFX_Text(pFX, "CourseIDParam",
    m_strCourseIDParam);
```

`DoFieldExchange` recognizes two kinds of fields: columns and parameters. The call to the **CDaoFieldExchange** member function **SetFieldType** indicates what kind of field(s) follow in the DFX function calls. In this example, there is one parameter: `m_strCourseIDParam`.

The name of the column for the parameter in the **DFX_Text** call — "CourseIDParam" — is arbitrary; you can provide any name you want.

3 Save your work.

Note No code is added during the next two procedures; the code already exists in the previous code block.

▶ To specify a parameterized filter

- Before the call to the base class function **CDaoRecordset::Open**, which is called by **CDaoRecordView::OnInitialUpdate**, specify the parameterized filter, as shown in this line (which you've already added):

```
m_pSet->m_strFilter = "CourseID = CourseIDParam";
```

`CourseID` is a column (field) name, and `CourseIDParam` is a named parameter associated with the column. Its value will be substituted at run time. DAO parameters are always named, rather than positional, as in most ODBC code. If you have more than one parameter in **m_strFilter**, such as:

```
m_pSet->m_strFilter = "CourseID = CourseIDParam AND SectionNo = SectionNoParam";
```

you must make multiple RFX calls after the call to:

```
pFX->SetFieldType(CDaoFieldExchange::param);
```

▶ To supply the run-time parameter value

- Assign the value to the previously bound parameter data member, as shown in the following line (which you've already added in the **OnInitialUpdate** function).

```
m_pSet->m_strCourseIDParam =
    pDoc->m_courseSet.m_CourseID;
```

This sets the parameter value to be the first course record retrieved from the `CCourseSet` recordset. All parameter values must be assigned before calling **CDaoRecordset::Open** (or **CDaoRecordView::OnInitialUpdate**) or, as you will see later, before calling **CDaoRecordset::Requery**.

Note The technique just described for parameterizing a query is not the only approach available. The MFC DAO classes provide two alternative ways to manage recordsets, including any parameters you might give them. The approach described here relies on the wizards and uses the DAO record field exchange (DFX) mechanism to move data between the database and the recordset and to manage parameters. The alternative approach uses a different mechanism, called dynamic binding, which is described in the article "DAO Queries: Filtering and Parameterizing Queries" in *Programming with MFC*.

Finishing DaoEnrol Step 2

Use the remainder of the instructions in the Enroll tutorial to complete the DaoEnrol tutorial. Begin with the section titled "Reusing a Database Object Opened by Another Recordset" in Chapter 32. At the end of Step 2, compile and run the application.

DaoEnrol Step 3

In Chapter 33, "Adding and Deleting Records," you will learn how to add and delete records. Follow the sections in Chapter 33 and the substitutions listed below to complete Step 2 of DaoEnrol.

In Chapter 33:

- "Adding and Deleting Records"
- "About Step 3"
- "Creating the Step 3 User Interface"
- "Add an Accelerator for the Refresh Command"
- "Create Handlers for Add, Refresh, and Delete"
- "The Basics of Adding, Editing, and Deleting Records"
- "Implementing the Add Command"

Next, in Chapter 34:

- "Updating the Data Source with the Added Record in DaoEnrol"

  The steps for updating a data source are different for a DAO-based database application.
- "Disabling Combo Box Logic in Add Mode in DaoEnrol"

  The instructions here are identical, but are provided for your convenience.
- "Implementing the Delete Command in DaoEnrol"

  Exception handling differs in the DAO database classes, so an appropriate code block is provided in this section.
- "Implementing the Refresh Command in DaoEnrol"

  The Move operations in **CDaoRecordSet** are slightly different than those in **CRecordset**.

Note The figures in Chapter 33 show "Enroll" in the title bar of the mainframe window. Your application will show "DaoEnrol". Remember to substitute **CDaoRecordset** for **CRecordset** and **CDaoRecordView** for **CRecordView** as appropriate.

Updating the Data Source with the Added Record in DaoEnrol

Note You should have completed the sections "Adding and Deleting Records" through "Implementing the Add Command" in Chapter 33 before starting this section.

Add mode is completed when the user moves off the record. DaoEnrol implements this by overriding the **CDaoRecordView::OnMove** member function.

▶ To implement Add functionality in the OnMove function override

1 With SECTFORM.CPP open in the text editor, select class CSectionForm in the WizardBar Object IDs drop-list.

2 In the Messages drop-list, select OnMove, and choose Yes when prompted to create a handler.

3 Fill in the skeleton `OnMove` function with the following code:

```
if (m_bAddMode)
{
    if (!UpdateData())
        return FALSE;
    try
    {
        m_pSet->Update();
    }
    catch(CDaoException* e)
    {
        AfxMessageBox(e->
            m_pErrorInfo->m_strDescription);
        e->Delete();
    }
    m_pSet->Requery();
    UpdateData(FALSE);
    m_ctlSection.SetReadOnly(TRUE);
    m_bAddMode = FALSE;
    return TRUE;
}
else
    AfxMessageBox(e->
        m_pErrorInfo->m_strDescription);
}
return FALSE;
```

Error handling in the DAO database classes differs from that of the ODBC database classes. The implementation of **else** in this code block requires a message box with error information to complete the **if/else** control structure.

In its default **CDaoRecordView** implementation, **OnMove** moves to the next, previous, first, or last record. If the application has changed the recordset field data members for the current record before the move, the framework updates the data source before moving to another record.

Normally, the **Move** commands behave as you might expect: **MoveNext** moves to the next record, and so on. But as a consequence of the decision to **Requery** during the **Add** operation, when the user chooses any move command when adding a record,

DaoEnrol always effectively moves to the first record. That's because requerying the recordset automatically sets the recordset to the first record.

Disabling Combo Box Logic in Add Mode in DaoEnrol

Step 2 implemented a handler for selecting a course in the combo box. The handler requeried the parameterized `CSectionSet` for the newly selected course. In Step 3, the combo box takes on the additional duty of allowing the user to specify the course for a new section record being added. During add mode, you don't want to requery the recordset when the user selects a course from the combo box. Therefore, you need to put the requery logic inside an **if** clause that is executed only if add mode isn't in effect.

▶ To disable normal combo box logic while in add mode

1 Use ClassView to jump to the `OnSelendokCourseList` handler in class `CSectionForm`.

2 Place an **if** block around the requery code in the `OnSelendokCourseList` handler, so the handler now appears as follows:

```
void CSectionForm::OnSelendokCourselist()
{
if (!m_pSet->IsOpen() )
        return;
    m_ctlCourseList.GetLBText(m_ctlCourseList.GetCurSel(),
        m_pSet->m_strCourseIDParam);
    if (!m_bAddMode)
    {
        m_pSet->Requery();
        if (m_pSet->IsEOF())
        {
            m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
            m_pSet->m_CourseID = m_pSet->m_strCourseIDParam;
        }
        UpdateData(FALSE);
    }
}
```

Implementing the Delete Command in DaoEnrol

In response to a Delete command, the record view deletes the current record by calling the `Delete` member function of its associated recordset.

▶ To implement the Delete command

1 Use ClassView to jump to the `OnRecordDelete` skeleton function in class `CSectionForm`.

2 Implement the handler with the following code:

```
try
{
    m_pSet->Delete();
}
catch(CDaoException* e)
{
    AfxMessageBox(e->
        m_pErrorInfo->m_strDescription);
    e->Delete();
}
// Move to the next record after the one just deleted
m_pSet->MoveNext();
// If we moved off the end of file, move back to last record
if (m_pSet->IsEOF())
    m_pSet->MoveLast();
// If the recordset is now empty, clear the fields left over
// from the deleted record
if (m_pSet->IsBOF())
    m_pSet->SetFieldNull(NULL);
UpdateData(FALSE);
```

Catch any exceptions thrown by the recordset's `Delete` function so that errors are reported to the user. The **CDaoException** data member **m_pErrorInfo** retrieves fairly user-friendly error messages prepared by the underlying **DAOException** object.

For DaoEnrol, the decision was made to move to the record following the deleted record. You could move to the previous record after a delete operation or anywhere else as long as you, or the user, moves off the deleted record.

Implementing the Refresh Command in DaoEnrol

The Refresh command cancels add mode, if the user had previously chosen Add, or it discards any changes the user may have made on the form for the current record. In the first case, DaoEnrol cancels add mode by calling:

```
CDaoRecordset::Move(0);
```

When you call **AddNew** to begin the add operation, the framework stores a copy of the current record's fields before allowing the user to enter new values in the record view's controls. Calling **Move** as shown here "refreshes" the current record — and effectively cancels the add operation. It restores the record that was current before add mode began. This also works if you called **Edit** instead of **AddNew**.

When the user cancels add mode, DaoEnrol makes the Section control read-only again, for reasons explained earlier.

▶ To implement the Refresh command

1 Use ClassView to jump to the `OnRecordRefresh` skeleton handler in class `CSectionForm`.

2 Implement the handler function with the following code:

```
if (m_bAddMode)
{
    m_pSet->CancelUpdate();
    m_pSet->Move(0);
    m_ctlSection.SetReadOnly(TRUE);
    m_bAddMode = FALSE;
}
// Copy fields from recordset to form, thus
// overwriting any changes the user may have made
// on the form
UpdateData(FALSE);
```

You are now ready to build and run Step 3 of DaoEnrol. Try the new Add, Refresh, and Delete commands. Try forcing the two exceptions handled by DaoEnrol — deleting a section that has Enrollment records, or to adding a duplicate section.

This concludes the DaoEnrol tutorial.

DaoEnrol Step 4: The DAOENROL Sample

The DAOENROL sample is Step 4 of the DaoEnrol tutorial, but this step is not documented, nor is separate source code provided for each of the DaoEnrol tutorial steps.

The ENROLL sample, which is Step 4 of the Enroll tutorial, is also not documented, but the code is very similar to that of DAOENROL. You may find it useful to compare the source code for the two samples to understand some of the differences in the implementation of the ODBC and DAO database classes.

PART 5

Windows 95 Compliance

CHAPTER 35

Adding Windows 95 Functionality

Microsoft Visual C++ 4.0 and the Microsoft Foundation Class Library are designed to help you write applications for Windows 95. Several of the requirements for the Windows 95 logo are satisifed automatically by any MFC application built using Visual C++. Fulfilling the remaining requirements, such as OLE support, requires knowledge of the specific nature of your application, making it impossible for MFC to generate a completely Windows 95-logo compliant application automatically. However, MFC provides classes containing all the generic code needed, so that all you have to do is add the code specific to your application. This chapter describes the process of meeting the logo requirements for a sample MFC application.

Note This chapter is not a substitute for the Windows 95 Logo Criteria document. To get a copy of the Logo Criteria document and related documents, see the section "For More Information on the Windows 95 Logo."

Summary of the Logo Requirements

There are five basic requirements your application must meet to qualify for the Windows 95 logo, no matter what type of application it is:

- Win32 executable

  Your application must be a Win32 executable using the PE (Portable Executable) format. This requirement is satisfied automatically, because Visual C++ for the Intel platform always produces executables in the PE format.

  If you have a 16-bit application that you need to port, see "Porting 16-Bit Code to 32-Bit Windows" in *Programming Techniques*.

- UI and Shell support

  Your application must follow the Windows 95 application setup guidelines, register large and small icons, and use system color and metrics. It's also recommended that your application provide context menus through the right mouse button, use the common dialogs and controls, and follow the user-interface

guidelines set forth in the *Windows Interface Guidelines for Software Design* on MSDN, in the "SDK" section of the "Product Documentation" category.

The section "Following UI Recommendations" below discusses how to add this support using MFC.

- Windows NT compatibility

  Your application must run successfully on both Windows 95 and Windows NT 3.5 (or greater). If your application uses functionality specific to either operating system, its behavior must degrade gracefully when run on the other operating system (that is, an execution error should not result).

  Meeting this requirement depends on your use of API functions specific to Windows 95 or Windows NT. For a list of these APIs, see the paper "Diving into the Requirements for the Windows 95 Logo" on MSDN, in the "Operating Systems" category of the "Backgrounders and White Papers" section.

- Long filename support

  Your application must be able to accept and store long filenames and display them in its title bar, in dialogs and controls, and so on.

  The MFC library supports the use of long filenames, so you can pass a long filename to any MFC function taking a filename as a parameter. You should also use the MFC common dialog class **CFileDialog** when requesting a filename from the user. Make sure that any filename-handling code of your own can handle filenames longer than eight characters and filenames containing spaces or other special characters. For more information on long filenames, see "Supporting Long Filenames" on MSDN, in the "Programming the Windows 95 User Interface" volume of the "Books and Periodicals" section.

- Plug and Play support

  This is recommended, but not required. Your application should respond appropriately to the **WM_DEVICECHANGE**, **WM_DISPLAYCHANGE**, and **WM_POWERBROADCAST** messages, which signal changes in peripheral devices, the display resolution, or the system power status, respectively.

  The MFC library responds to the **WM_DISPLAYCHANGE** message and resizes windows and toolbars according to the new system metrics. How an application should respond to the **WM_DEVICECHANGE** and **WM_POWERBROADCAST** messages depends on the specific application; for an example of the most common response to these messages, see the Plug-and-Play awareness component in the Component Gallery.

There are also three other requirements your application must meet if it is file-based (that is, if your application's primary purpose is to create, edit, and save files):

- UNC path support

  Your application must support Universal Naming Convention (UNC) paths. That means your application must be able to accept and store paths of the form "\\server\share\directory" directly, without requiring the user to assign a drive letter to the server beforehand.

  The MFC library supports the use of UNC paths, so you can pass a UNC path to any MFC function taking a pathname as a parameter. You should also use the MFC common dialog class **CFileDialog** when requesting a filename from the user. Make sure that any pathname-handling code that you write can handle double backslashes instead of a drive letter at the beginning of a path.

- OLE support

  Your application must either be an OLE container or an OLE server, or both. In addition, if it's an OLE container, it must act as a target for drag-and-drop operations, and if it's an OLE server, it must act as a source for drag-and-drop operations. It's also recommended (thought not required) that you support OLE Automation, and provide summary information with your documents.

  The section "Adding OLE Support" later in this chapter discusses how to add this functionality using MFC.

- MAPI support

  Your application must include a Send Mail command on the File menu to enable the user to send the current document as a piece of mail, using MAPI or the Common Messaging Call (CMC) API.

  The section "Adding MAPI Support" later in this chapter discusses how to add this functionality using MFC.

There are certain additions and exceptions to the previous requirements, depending on the type of application you're developing; these additions and exceptions are described in the Logo Criteria document.

The remainder of this chapter describes the changes made to the DRAWCLI sample application to make it meet the Windows 95 logo requirements, as well as certain user-interface recommendations; you can use this chapter as a guide for modifying your own MFC application.

Following UI Recommendations

Your Windows 95 application must define both large and small icons. AppWizard automatically defines default 32x32 and 16x16 icons for both the application and the document type when generating MFC applications. If you replace the default icons with your own, remember to replace both the large and small versions. In addition, you have to register the icon for your document type with the system registry under the **DefaultIcon** key. MFC does this with the **CWinApp::RegisterShellFileTypes** function; AppWizard inserts a call this function automatically if you specify a three-letter filetype extension for your application's document. You can also register the icon in your setup application; see "Creating a Setup and an Uninstall Program" below for more information.

Your Windows 95 application must use the system colors and metrics for its dialogs to be consistent with the user's settings. The MFC library calls the **GetSysColor** and **GetSystemMetrics** API functions to determine how dialogs and controls should be displayed. As a result, your MFC application's appearance will automatically reflect the system colors and metrics, even if the user changes some attribute while the application is running.

It's also recommended, though not required, that your Windows 95 application use the Windows common dialogs where applicable. The common dialogs provide a great deal of functionality in their support of long filenames and UNC pathnames, as mentioned above. An application that uses the MFC framework automatically uses the common file and print dialogs, and if you want to use any of the other common dialogs, you can use the MFC library's wrappers for those dialogs. See Technical Note 60 and the descriptions of **CFileDialog**, **CPrintDialog**, **CFindReplaceDialog**, **CColorDialog**, and **CFontDialog** in the *Class Library Reference*.

Using Tabbed Property Pages

It's recommended that your Windows 95 application use tabbed property pages where applicable. Tabbed property pages are useful for dialogs whose purpose is to let the user modify the attributes of some object on the screen. MFC provides support for property pages through the classes **CPropertySheet** and **CPropertyPage**.

The DRAWCLI sample previously used a dialog box to allow the user to set the fill mode and the pen size. For consistency with the Windows 95 user-interface guidelines, DRAWCLI now uses a property page to do so.

You can use the Component Gallery to add a property sheet to your application. However, for DRAWCLI, the task is not adding a property sheet, but converting an existing dialog into a property sheet. Depending on your application's needs, you can use Component Gallery to add property sheets and modify them as needed, or you can add them manually.

If you're converting an existing dialog into a property page, use the following procedure:

- From within Visual C++'s dialog editor, do the following:
  - Check the Disabled and Titlebar properties for the dialog.
  - Choose the Child style and Thin borders.
  - Optional: Modify the caption for the dialog. This caption will now appear on the tab for the property page instead of in the dialog's title bar.
- Change the base class of your dialog class from **CDialog** to **CPropertyPage** and modify the constructor for your dialog class to call **CPropertyPage**'s constructor in its base-initializer list.

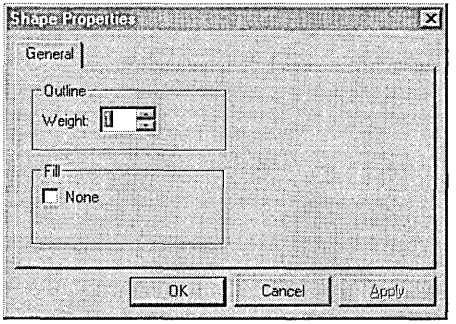
To display the property page, attach it to a **CPropertySheet** object and call **DoModal** on the **CPropertySheet** object. Here's how DRAWCLI does it, in its implementation of `CDrawObj::OnOpen`:

```
CPropertySheet sheet( "Shape Properties" );
CRectDlg dlg; // derived from CPropertyPage

dlg.m_bNoFill = !m_bBrush;
dlg.m_penSize = m_bPen ? m_logpen.lopnWidth.x : 0;
sheet.AddPage( &dlg );

if (sheet.DoModal() != IDOK)
    return;

m_bBrush = !dlg.m_bNoFill;
m_bPen = dlg.m_penSize > 0;
if (m_bPen)
{
    m_logpen.lopnWidth.x = dlg.m_penSize;
    m_logpen.lopnWidth.y = dlg.m_penSize;
}
```

The string passed to the constructor for **CPropertySheet** is used as the title in the dialog's title bar. Before calling **DoModal**, the current values are copied into the property sheet. After **DoModal** returns, the new values are copied back.

Figure 35.1 shows what DRAWCLI's property page looks like. For more information on implementing property pages with MFC, see "Property Sheets" in *Programming with MFC* and the descriptions of **CPropertyPage** and **CPropertySheet** in the *Class Library Reference*. For a demonstration of different types of property sheets, see the PROPDLG sample application in the MFC samples.

Figure 35.1 DRAWCLI's New Property Sheet



Using Common Controls

It's recommended that your Windows 95 application use the common controls where applicable. Visual C++'s dialog editor offers the common controls on its control palette, and MFC provides classes for all the Windows common controls

One place in the DRAWCLI sample where a common control can be used is in the dialog for specifying the thickness of the drawing pen. DRAWCLI formerly used a simple edit control in which the user typed the pen thickness; now DRAWCLI adds a spin control, allowing the user to use the mouse or the arrow keys to increment or decrement the pen thickness.

You can add a spin control to your application by using the Visual C++ dialog editor; associate the spin control with an edit control by checking the Auto Buddy and Set Buddy Integer properties on the control's property page. Next, use the Member Variables tab of the ClassWizard dialog to add a member variable for the spin control. This causes ClassWizard to do two things:

- Declare a member of type **CSpinButtonCtrl** in the dialog class.
- Add a call to **DDX_Control** in the dialog's **DoDataExchange** member function, associating the spin control with the member variable.

You must manually add the statement `#include <afxcmn.h>` to your STDAFX.H file to include MFC's common control support in an existing application. If you generate a new application with AppWizard, AppWizard now inserts this statement into STDAFX.H automatically.

Finally, override the **OnInitDialog** member function of the dialog class to initialize the spin control. Here's what the function looks like in DRAWCLI:

```
BOOL CRectDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    m_SpinCtrl.SetRange(0, 100);
    m_SpinCtrl.SetBase(10);
    m_SpinCtrl.SetPos(1);
    return TRUE;
}
```

This function sets the initial, minimum, and maximum values for the spin control. It also sets the spin control to use decimal notation.

For more information on using MFC's common control classes, see "MFC: Windows 95 Support" in *Programming with MFC* and the descriptions of **CAnimateCtrl**, **CHeaderCtrl**, **CHotKeyCtrl**, **CListCtrl**, **CProgressCtrl**, **CSliderCtrl**, **CSpinButtonCtrl**, **CStatusBarCtrl**, **CTabCtrl**, **CToolBarCtrl**, **CToolTipCtrl**, and **CTreeCtrl** in the *Class Library Reference*. For a demonstration of various common controls, see the CMNCTRLS and FIRE sample applications in the MFC samples.

Displaying a Shortcut Menu

It's recommended that your Windows 95 application use the right mouse button for displaying a shortcut menu, providing easy access to the most commonly used commands. What commands you offer on your shortcut menu will depend on whether the menu is invoked for a particular window, a particular object, and so on. See *Windows Interface Guidelines for Software Design* on MSDN for guidelines on what kind of commands you should offer.

You can use Component Gallery to add a shortcut menu that applies to a given view class in your application. However, DRAWCLI displays the shortcut menu only when the right button is clicked over a selected object. Depending on your own application's needs, you can use the shortcut menu offered by Component Gallery as-is, you can make modifications as required, or you can add one manually.

The DRAWCLI sample provides a shortcut menu containing several commands from the Edit menu.

To add a shortcut menu to your application, first use the Visual C++ menu editor to create a menu bar without a title, and then define the shortcut menu as the first menu.

To display the shortcut menu, your application needs a handler for the **WM_CONTEXTMENU** message. You can use ClassWizard to make the following changes to your source files:

- Add a declaration for an **OnContextMenu** member function in the declaration of your view class.
- Add an **ON_WM_CONTEXTMENU** macro in the message map for your view class.
- Add a skeletal definition for the **OnContextMenu** member function.

You then fill in the definition of **OnContextMenu**. Here's DRAWCLI's implementation of `CDrawView::OnContextMenu`:

```
void CDrawView::OnContextMenu(CWnd* /*pWnd*/, CPoint point)
{
    // make sure window is active
    GetParentFrame()->ActivateFrame();

    CPoint local = point;
    ScreenToClient(&local);
    ClientToDoc(local);

    CDrawObj* pObj;
    pObj = GetDocument()->ObjectAt(local);
    if (pObj != NULL)
    {
        if (!IsSelected(pObj))
            Select(pObj, FALSE);
        UpdateWindow();

        CMenu menu;
        if (menu.LoadMenu(ID_POPUP_MENU))
        {
            CMenu* pPopup = menu.GetSubMenu(0);
            ASSERT(pPopup != NULL);

            pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                point.x, point.y,
                AfxGetMainWnd()); // use main window for cmds
        }
    }
}
```

DRAWCLI displays a pop-up menu only when the right mouse button is clicked over an object, so this function first checks whether an object lies where the mouse event occurred, and if so, the function selects it. (**OnContextMenu** receives the location of the mouse event in screen coordinates, not in client coordinates like handlers such as **OnLButtonDown**. Accordingly, the function converts the coordinates using **CWnd::ScreenToClient** before doing hit-testing.) The function then loads the ID_POPUP_MENU menu and calls **CMenu::GetSubMenu** to get its first sub-menu. Finally the function calls **CMenu::TrackPopupMenu** on the sub-menu to display it as a pop-up.

Note that, as long as the pop-up menu commands are duplicates of commands you've defined elsewhere, there's no need to define any new **ON_COMMAND** or **ON_UPDATE_COMMAND_UI** macros.

You can also support shortcut menus in the Windows 95 shell. A shortcut menu that offers basic commands such as Delete and Rename is available for all files in the shell. Some commands, such as Open or Print, are enabled only if there is an application associated with the file's extension. You can enable these commands for

files created by your MFC application by calling **CWinApp::RegisterShellFileTypes**, passing **TRUE** as a parameter; the parameter is needed for shell registration to work properly under Windows 95. (AppWizard inserts a call to this automatically if you specify a filename extension for your files.) This function adds entries to the system registry that let the shell invoke your application to open or print a file. You can also write a shell-extension DLL that customizes the shortcut menu for your application's files; for more information, see "Context Menu Handlers" in the *Programmer's Guide to Windows 95* on MSDN.

Using the System Registry

Your Windows 95 application must not store any information in the WIN.INI and SYSTEM.INI files; instead, use the system registry to store any initialization information. MFC applications never store information in WIN.INI or SYSTEM.INI. Instead, they normally use a private .INI file to store information such as the MRU (most recently used) file list, the position of dockable toolbars, etc. Private .INI files are permissible for Windows 95 applications (though they are not recommended), as long as they are stored in the application directory and are deleted when the application is uninstalled.

You can make your MFC application compliant by using the **CWinApp::SetRegistryKey** function; this function makes your application use the registry instead of a private .INI file. Call this function from the **InitInstance** member function of your **CWinApp**-derived class, specifying the name of your software company (in string form) as the parameter; this creates a subkey in the registry (under HKEY_CURRENT_USER) with your software company's name. After this function is called, all the information that your application would normally write to its .INI file is instead stored in the registry under that key.

Creating a Setup and an Uninstall Program

Your Windows 95 application must be accompanied by a setup program having a graphical user interface. This program must be named SETUP.EXE so it can be identified by the Add/Remove Programs applet in the Control Panel. Your application must also add entries to the system registry specifying the command needed to uninstall your application. Again, this allows your uninstall program to be identified by the Add/Remove Programs applet.

A complete discussion of what's involved in writing your own setup and uninstall programs is beyond the scope of this chapter. For information on the requirements for setup and uninstall programs, see the article "Windows 95 Application Setup Guidelines for ISVs" on MSDN, in the "Windows Articles" section of the "Technical Articles" category.

However, as a shortcut for developing setup and uninstall programs, Visual C++ includes the SDK Edition of Stirling Software's InstallSHIELD. Using this toolkit, you can write a script that specifies the steps you want performed during installation; the InstallSHIELD SETUP application uses this script to install your product. The DRAWCLI sample includes a setup script for use with InstallSHIELD.

The script for DRAWCLI's setup program installs the executable for DRAWCLI, the DLLs needed by DRAWCLI, a help file, and some sample files. You can use the script as a basis for creating your own setup program using InstallSHIELD.

At a minimum, the changes you need to make to the script are as follows:

- Change the size requirements to reflect the sizes of your program files, help files, and sample files.
- Change the application name.
- Change the version number.
- Change the company name. (This must be consistent with the one you specify in your call to **CWinApp::SetRegistryKey**).
- Change the command lines for the program group items.
- Change the filetype extension. (This must be consistent with the extension you specified in AppWizard; this is used by **CWinApp::RegisterShellFileTypes**).
- Change the filetype name. (This must be consistent with the one generated by AppWizard for **CDocTemplate::regFileTypeId**; this is used by **CWinApp::RegisterShellFileTypes**).

Other changes may be necessary depending on the details of the installation process for your application.

With InstallSHIELD, you don't have to write a separate script for uninstallation. InstallSHIELD's uninstall program, UNINST, uses a log file that is created automatically during the setup process. This log file enables UNINST to reverse the steps performed during setup.

For more information on using InstallSHIELD, see the *InstallSHIELD User's Guide*. For more information on using the system registry, see "Integrating with the System" in the *Windows Interface Guidelines for Software Design* on MSDN.

For a list of the files that you can redistribute with your application, see "DLLs: Redistribution" in *Programming with MFC* or the document REDISTRB.WRI in the MSDEV\REDIST subdirectory.

Adding OLE Support

The MFC library defines numerous classes for supporting OLE functionality, including classes that implement the standard OLE dialog boxes. AppWizard also provides support for the automated creation of OLE container and OLE server applications. See the OLE tutorials in *Tutorials* for information on using MFC and AppWizard to write OLE applications. The section "Creating an OLE Server" describes how to make the SCRIBBLE sample an OLE server. The section "Creating an OLE Container" describes how to write an OLE container, using the CONTAINER sample as an example.

The previous version of the DRAWCLI sample already had OLE container functionality implemented, so it was not necessary to add this feature to meet Windows 95 logo requirements. However, DRAWCLI previously did not have drag-and-drop support; the following section describes the addition of that functionality.

For other examples of OLE containers, see the CONTAINER and OCLIENT sample applications in the MFC samples. For examples of OLE servers, see the HIERSVR sample application and Step 7 of the SCRIBBLE tutorial sample in the MFC samples. MFC Samples are found under Samples in Books Online.

Being a Drop Target

Because DRAWCLI is an OLE container, it needs drop target functionality to meet the Windows 95 requirements. (Conversely, an OLE server needs drop source functionality.)

MFC provides support for drop-target functionality through the **COleDropTarget** class and certain member functions in the **CView** class. Being a drop target involves the following tasks:

- Registering your window as a drop target.
- Checking a **COleDataObject** object to see if you can accept a drop operation from it.
- Providing target feedback for the user.
- Scrolling the document when the mouse cursor moves near the border of the document window.
- Performing a paste operation from a **COleDataObject** object.

Most of these tasks are application-specific, so MFC can't complete them for you. However, MFC does provide support for scrolling a document during a drag-and-drop operation, which is traditionally one of the more difficult tasks involved in being a drop target. The following sections describe how the other tasks are completed by the DRAWCLI sample.

One requirement for being a drop target was already satisfied by the previous version of DRAWCLI: the ability to perform a paste operation from a **COleDataObject** object. The previous version of DRAWCLI allowed OLE embedded objects to be inserted with the Paste command; the new version of DRAWCLI takes advantage of this fact by reusing the existing `CDrawView::PastEmbedded` function. If your container application doesn't support this use of the Paste command, you should add that before implementing drop-target functionality. For information on creating embedded objects with the Paste command, see "Data Objects and Data Sources (OLE)" and "Clipboard: Copying and Pasting Data" in *Programming with MFC*.

Registration

In order for DRAWCLI to be informed of drag-and-drop operations, it must register its windows as drop targets. This requires adding a member variable to `CDrawView`:

```
COleDropTarget m_dropTarget;
```

The **COleDropTarget** member is used to register the view as a drop target.

A DRAWCLI window should be registered as a drop target as soon as it's created. Consequently, DRAWCLI needs a handler for the **WM_CREATE** message, which requires three modifications to the source files:

- Adding a declaration for an **OnCreate** member function in the declaration of `CDrawView`.
- Adding an **ON_WM_CREATE** macro in the message map for `CDrawView`.
- Adding a definition for `CDrawView::OnCreate`.

You can use ClassWizard to provide the necessary declarations and provide a skeletal function definition.

Here's what the implementation of `CDrawView::OnCreate` looks like:

```
int CDrawView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CScrollView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // register drop target
    if( m_dropTarget.Register( this ) )
        return 0;
    else
        return -1;
}
```

This function calls **COleDropTarget::Register** on the `m_dropTarget` member variable. This informs the OLE system DLLs that DRAWCLI windows are willing to accept dragged objects.

The function **COleDropTarget::Revoke** is called automatically when the destructor is called.

Providing Target Feedback

A drop target must provide user feedback, that is, the visual indication to the user of how the window would respond to a drag-and-drop operation. For an application such as a word processor, this might consist of a shaded caret indicating where the dropped object would be inserted. For DRAWCLI, user feedback consists of a focus rectangle indicating the size and position of the object if it were to be dropped.

DRAWCLI declares some new member variables to manage this user feedback. The relevant declarations in `CDrawView` are as follows:

```
CPoint m_dragPoint;          // current position
CSize m_dragSize;            // size of dragged object
CSize m_dragOffset;          // offset of focus rect
DROPEFFECT m_prevDropEffect;
static CLIPFORMAT m_cfObjectDescriptor;
BOOL m_bDragDataAcceptable;

BOOL GetObjectInfo(COleDataObject* pDataObject,
                CSize* pSize, CSize* pOffset);
```

Some of the member variables store the size and position of the focus rectangle. The `m_bDragDataAcceptable` is a flag indicating whether usable data is available from the drag-and-drop operation. The `GetObjectInfo` function is a helper function described below.

Two of the member variables must be initialized. One of them is `m_prevDropEffect`, which gets initialized in the `CDrawView` constructor:

```
m_prevDropEffect = DROPEFFECT_NONE;
```

The other one is `m_cfObjectDescriptor`; this was declared as a static member, which means that it must be initialized at file scope, outside of the `CDrawView` constructor:

```
CLIPFORMAT CDrawView::m_cfObjectDescriptor =
     (CLIPFORMAT)::RegisterClipboardFormat(_T("Object Descriptor"));
```

DRAWCLI registers the string "Object Descriptor" so it can get the handle for the **CF_OBJECTDESCRIPTOR** Clipboard format defined by OLE. This handle is used below in the `CDrawView::GetObjectInfo` helper function.

Before DRAWCLI can draw this focus rectangle, it needs to know the size of the object. This is the purpose of the `GetObjectInfo` helper function; this function queries a data object for the **CF_OBJECTDESCRIPTOR** clipboard format:

```
BOOL CDrawView::GetObjectInfo(COleDataObject* pDataObject,
    CSize* pSize, CSize* pOffset)
{
    ASSERT(pSize != NULL);

    // get object descriptor data
    HGLOBAL hObjDesc =
                pDataObject->GetGlobalData(m_cfObjectDescriptor);
```

```
    if (hObjDesc == NULL)
    {
        if (pOffset != NULL)
            *pOffset = CSize(0, 0); // fill in defaults instead
        *pSize = CSize(0, 0);
        return FALSE;
    }
    ASSERT(hObjDesc != NULL);

    // else, got CF_OBJECTDESCRIPTOR. Lock it down and extract size.
    LPOBJECTDESCRIPTOR pObjDesc =
                    (LPOBJECTDESCRIPTOR)GlobalLock(hObjDesc);
    ASSERT(pObjDesc != NULL);
    pSize->cx = (int)pObjDesc->sizel.cx;
    pSize->cy = (int)pObjDesc->sizel.cy;
    if (pOffset != NULL)
    {
        pOffset->cx = (int)pObjDesc->pointl.x;
        pOffset->cy = (int)pObjDesc->pointl.y;
    }
    GlobalUnlock(hObjDesc);
    GlobalFree(hObjDesc);

    // successfully retrieved pSize & pOffset info
    return TRUE;
}
```

This function calls **COleDataObject::GetGlobalData** and acquires the **CF_OBJECTDESCRIPTOR** data in a block of global memory. The function locks the memory down, reads the size attributes, and then unlocks and frees the block of memory. This function is used by **OnDragEnter** in the code samples in "Handling a Drag-and-Drop Operation," following.

Handling a Drag-and-Drop Operation

The actual work of handling a drag-and-drop operation is done by overriding four member functions defined by **CView**:

```
virtual BOOL OnDrop(COleDataObject* pDataObject,
            DROPEFFECT dropEffect, CPoint point);
virtual DROPEFFECT OnDragEnter(COleDataObject* pDataObject,
                DWORD grfKeyState, CPoint point);
virtual DROPEFFECT OnDragOver(COleDataObject* pDataObject,
                DWORD grfKeyState, CPoint point);
virtual void OnDragLeave();
```

These functions are called by the OLE system DLL (via the MFC framework) during a drag-and-drop operation.

The first function called during an actual drag-and-drop operation is **OnDragEnter**, which is called when the mouse first enters the window:

```
DROPEFFECT CDrawView::OnDragEnter(COleDataObject* pDataObject,
    DWORD grfKeyState, CPoint point)
```

```
{
    ASSERT(m_prevDropEffect == DROPEFFECT_NONE);
    m_bDragDataAcceptable = FALSE;
    if (!COleClientItem::CanCreateFromData( pDataObject ))
        return DROPEFFECT_NONE;

    GetObjectInfo(pDataObject, &m_dragSize, &m_dragOffset);
    CClientDC dc(NULL);
    dc.HIMETRICtoDP(&m_dragSize);
    dc.HIMETRICtoDP(&m_dragOffset);

    return OnDragOver(pDataObject, grfKeyState, point);
}
```

This function first checks whether the **COleDataObject** provided by the drop source contains data that DRAWCLI can use. If not, the function sets a flag indicating that the data is unacceptable, and returns **DROPEFFECT_NONE**, indicating that a drop operation would have no effect. On the other hand, if DRAWCLI can use the data, then the function computes the size and position of the focus rectangle. It does this by using the `GetObjectInfo` helper function to get the size of the object in the **COleDataObject** object.

The next function called during a drag-and-drop operation is **OnDragOver**, which is called whenever the mouse moves within the window. This function is responsible for determining whether the window can accept the drop operation, and if so, for providing target feedback:

```
DROPEFFECT CDrawView::OnDragOver(COleDataObject*,
    DWORD grfKeyState, CPoint point)
{
    if (m_bDragDataAcceptable == FALSE)
        return DROPEFFECT_NONE;

    point -= m_dragOffset;  // adjust target rect by cursor offset

    // check for point outside logical area
    // GetTotalSize() returns the size passed to SetScrollSizes()
    CRect rectScroll(CPoint(0, 0), GetTotalSize());

    CRect rectItem(point,m_dragSize);
    rectItem.OffsetRect(GetDeviceScrollPosition());

    DROPEFFECT de = DROPEFFECT_NONE;
    CRect rectTemp;
    if (rectTemp.IntersectRect(rectScroll, rectItem))
    {
        // check for force link
        if ((grfKeyState & (MK_CONTROL|MK_SHIFT)) ==
            (MK_CONTROL|MK_SHIFT))
            de = DROPEFFECT_NONE; // we don't support linking
        // check for force copy
        else if ((grfKeyState & MK_CONTROL) == MK_CONTROL)
            de = DROPEFFECT_COPY;
```

```
        // check for force move
        else if ((grfKeyState & MK_ALT) == MK_ALT)
            de = DROPEFFECT_MOVE;
        // default -- recommended action is move
        else
            de = DROPEFFECT_MOVE;
    }

    if (point == m_dragPoint)
        return de;

    // else, cursor has moved -- need to update the drag feedback
    CClientDC dc(this);
    if (m_prevDropEffect != DROPEFFECT_NONE)
    {
        // erase previous focus rect
        dc.DrawFocusRect(CRect(m_dragPoint, m_dragSize));
    }
    m_prevDropEffect = de;
    if (m_prevDropEffect != DROPEFFECT_NONE)
    {
        m_dragPoint = point;
        dc.DrawFocusRect(CRect(point, m_dragSize));
    }
    return de;
}
```

This function first checks the flag `m_bDragDataAcceptable` to see if it's necessary to do any more processing. If so, the function then checks which, if any, keys are being depressed, determining whether the user wants a link operation, a move, or a copy. Since DRAWCLI isn't a linking container, this function returns **DROPEFFECT_NONE** when a link operation is specified, meaning that the view won't accept the dragged object. DRAWCLI does accept copy or move operations, so the function returns **DROPEFFECT_COPY** and **DROPEFFECT_MOVE** in those instances. (The drop source receives these **DROPEFFECT** values and modifies the mouse cursor appropriately.) Finally, if the operation is a copy or a move, the function draws the focus rectangle to indicate where the object would land if it were dropped.

If the mouse leaves the window without having dropped the object, the function that gets called is **OnDragLeave**. This function simply performs a little clean-up:

```
void CDrawView::OnDragLeave()
{
    CClientDC dc(this);
    if (m_prevDropEffect != DROPEFFECT_NONE)
    {
        // erase previous focus rect
        dc.DrawFocusRect(CRect(m_dragPoint,m_dragSize));
        m_prevDropEffect = DROPEFFECT_NONE;
    }
}
```

If the drag-and-drop operation was one that DRAWCLI was willing to accept, the function removes the target feedback by erasing the last focus rectangle drawn.

Finally, the function that gets called if the user actually performs the drop is **OnDrop**:

```
BOOL CDrawView::OnDrop(COleDataObject* pDataObject,
    DROPEFFECT dropEffect, CPoint point)
{
    ASSERT_VALID(this);

    // clean up focus rect
    OnDragLeave();

    // offset point as appropriate for dragging
    GetObjectInfo(pDataObject, &m_dragSize, &m_dragOffset);
    CClientDC dc(NULL);
    dc.HIMETRICtoDP(&m_dragSize);
    dc.HIMETRICtoDP(&m_dragOffset);
    point -= m_dragOffset;

    // invalidate current selection since it will be deselected
    OnUpdate(NULL, HINT_UPDATE_SELECTION, NULL);
    m_selection.RemoveAll();
    if (m_bDragDataAcceptable)
        PasteEmbedded( *pDataObject, point );

    // update the document and all views
    GetDocument()->SetModifiedFlag();
    GetDocument()->UpdateAllViews(NULL, 0, NULL);

    return TRUE;
}
```

This function determines the point at which the dropped object resides, deselects the currently selected object, and creates an OLE embedded object using the **COleDataObject** object provided by the drop source. Note that the `CDrawView::PasteEmbedded` function now takes an additional parameter compared with the previous version of DRAWCLI; this parameter lets the caller specify the location of a new embedded object, something that is unnecessary for Paste operations but is useful for drag-and-drop operations.

It would also be possible to make DRAWCLI a drop source, allowing the user to drag a selection from one of DRAWCLI's windows into another application's. However, DRAWCLI doesn't offer any common Clipboard formats, nor is DRAWCLI an OLE server, so there are no applications that could accept a dragged object originating from DRAWCLI. Consequently, the current version of DRAWCLI would not be a useful drop source.

For more information on MFC's drag-and-drop support, see "Drag and Drop (OLE)" in *Programming with MFC*. For more information about drag-and-drop in general, see *Inside OLE* or the OLE documentation on MSDN.

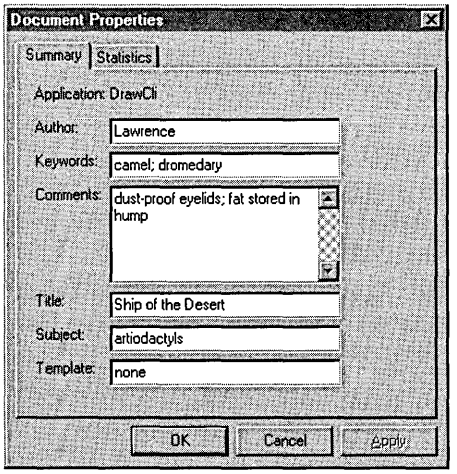
Providing Summary Information

It's recommended that, as part of its OLE support, your Windows 95 application store Summary Information with each file. This means offering a Summary Info command on your File menu, allowing the user to associate a title, keywords, or other attributes with a document. This also means using the compound file format (by calling **COleDocument::EnableCompoundFile** in the constructor for your document class) and writing these attributes as an OLE property set into a stream named "\005Summary Information" off the root storage of your document's compound file. A Windows 95 user can view the Summary Information when examining the properties of a file from within the Explorer.

MFC does not currently provide classes that manage Summary Information. However, the DRAWCLI application does include a sample implementation, in the form of the class `CSummInfo`, which you can use as an example when implementing your own; this class is used by the document class `CDrawDoc`. DRAWCLI also include property pages for displaying and modifying Summary Information; figure 35.2 shows what the interface for using Summary Information looks like.

For more information on OLE property sets in general and Summary Information in particular, see the article "OLE 2.0 Property Sets Exposed" on MSDN (in the "Windows Articles" section of the "Technical Articles" category) or the appendix "OLE Property Sets" in the OLE documentation.

Figure 35.2 DRAWCLI's Summary Information Property Sheet



Adding MAPI Support

Your Windows 95 application must include a Send Mail command on its File menu. The MFC library supplies an implementation of the Send Mail command in the form of two member functions of the **CDocument** class: **OnFileSendMail** and **OnUpdateFileSendMail**.

The **OnFileSendMail** member function saves the current document as an attachment to a mail message, and then invokes the mail client installed on the user's machine; the mail client allows the user to address the mail, add text, and then send the message. The **OnUpdateFileSendMail** member function enables or disables the Send Mail command depending on whether MAPI support is present on the user's machine.

Binding these functions to a menu item in DRAWCLI involves the following modifications to the source files:

- Adding a menu item to the File menu with the command ID **ID_FILE_SEND_MAIL**.
- Adding the following macros to the message map for `CDrawDoc`:

  ```
  ON_COMMAND(ID_FILE_SEND_MAIL, OnFileSendMail)
  ON_UPDATE_COMMAND_UI(ID_FILE_SEND_MAIL, OnUpdateFileSendMail)
  ```

  These macros go outside of the special `//{{AFX` delimeter comments because they're entered manually instead of through ClassWizard.

There's no need to override these functions; their definitions in the base class are invoked if the derived class doesn't provide new ones. For `CDrawDoc`, the **COleDocument** definitions of these functions get invoked. For a non-OLE document class, the **CDocument** definitions of these functions get invoked.

If you're creating a new MFC application, you can have AppWizard add the entries listed above by checking the Add MAPI Support checkbox.

For more information, see the article "MAPI Support" in MFC in *Programming with MFC*.

For More Information on the Windows 95 Logo

For more information on meeting the Windows 95 logo requirements, see the paper "Diving into the Requirements for the Windows 95 Logo" on MSDN, in the "Operating Systems" category of the "Backgrounders and White Papers" section.

There are a few special requirements for utilities and development tools. For details, please see the document on the Windows 95 Logo Technical Criteria, available in the locations listed below. In general, these locations will always include the most up-to-date information available on the Windows 95 logo:

- On the Internet use ftp or the World-Wide-Web (ftp://ftp.microsoft.com/PerOpSys/Win_News, http://www.microsoft.com).
- On The Microsoft Network, open Computers and Software, Software Companies, Microsoft, Windows 95, WinNews.
- On CompuServe®, type GO WINNEWS.
- On Prodigy™, JUMP WINNEWS.
- On America Online®, use keyword WINNEWS.
- On GEnie™, download files from the WinNews area under the Windows RTC.

You can access the Microsoft Developer Solutions Phone-Fax service by calling (800) 426-9400. Choose option 2 for Developer Solutions, then option 1 for the Faxback service; or dial (206) 635-2222. You can request a complete index of available documents. Documents on the Windows Logo are numbered in the 130s range.

You can also get answers to logo-related questions or request a pre-testing kit from the Windows Logo Department:

by email:	"winlogo@microsoft.com"
by fax:	(206) 936-7329, Attn: Windows Logo Department
by phone:	(206) 936-8220
by mail:	Microsoft Corporation Attn: Windows Logo Department, Bldg. 20 Redmond, WA 98052-6399

PART 6

Appendix

APPENDIX A

Accessibility for People with Disabilities

Microsoft is committed to making its products and services easier for everyone to use. This appendix provides information about the following features, products and services, which make Microsoft Windows, Microsoft Windows NT, and Microsoft Visual C++ more accessible for people with disabilities:

- Microsoft services for people who are deaf or hard-of-hearing
- Access Packs for either Microsoft Windows or Microsoft Windows NT, a software utility that makes using Windows or Windows NT easier for people with motion or hearing disabilities
- Keyboard layouts designed for people who type with one hand or a wand
- Microsoft software documentation on audio cassette, floppy disk, or compact disc (CD)
- Third-party utilities to enhance accessibility
- Hints for customizing Microsoft Windows or Microsoft Windows NT
- Other products and services for people with disabilities

Note The information in this section applies only to users who purchased Microsoft products in the United States. If you purchased Windows or Windows NT outside the United States, your package contains a subsidiary information card listing Microsoft support services telephone numbers and addresses. You can contact your subsidiary to find out whether the type of products and services described in this appendix are available in your area.

Microsoft Services for People Who Are Deaf or Hard-of-Hearing

Through a text telephone (TT/TDD) service, Microsoft provides people who are deaf or hard-of-hearing with complete access to Microsoft product and customer services.

You can contact Microsoft Sales Information Center on a text telephone by dialing (800) 892-5234 between 6:30 A.M. and 5:30 P.M. Pacific time. For technical assistance in the United States, you can contact Microsoft Support Network on a text telephone at (206) 635-4948 between 6:00 A.M. and 6:00 P.M. Pacific time, Monday through Friday, excluding holidays. In Canada, dial (905) 568-9641 between 8:00 A.M. and 8:00 P.M. Eastern time, Monday through Friday, excluding holidays. Microsoft support services are subject to Microsoft prices, terms, and conditions in place at the time the service is used.

Access Packs for Microsoft Windows and Microsoft Windows NT

Microsoft distributes Access Packs for Microsoft Windows and Microsoft Windows NT, which provide people with motion or hearing disabilities better access to computers running Windows or Windows NT. (If you are running Microsoft Windows 95, these same Access Pack features are already built in. See online Help for more information.) Microsoft Windows and Microsoft Windows NT Access Packs contain several features that:

- Allow single-finger typing of SHIFT, CTRL, and ALT key combinations.
- Ignore accidental keystrokes.
- Adjust the rate at which a character is repeated when you hold down a key, or turn off character repeating entirely.
- Prevent extra characters if you unintentionally press a key more than once.
- Enable you to control the mouse cursor by using the keyboard.
- Enable you to control the computer keyboard and mouse by using an alternate input device.
- Provide a visual cue when the computer beeps or makes other sounds.

Access Pack for Microsoft Windows is included on the Microsoft Windows Driver Library in the file ACCP.EXE. Access Pack for Microsoft Windows NT is included in the Microsoft Application Note WNO789. If you have a modem, you can download ACCP.EXE or WNO789.EXE, which are self-extracting archive files, from the following network services:

- CompuServe®
- GEnie™
- The Microsoft Network
- Microsoft Download Service (MSDL), which you can reach by calling (206) 936-6735 any time except between 1:00 A.M. and 2:30 A.M. Pacific time. Use the following communications settings:

For this setting	Specify
Baud rate	1200, 2400, 9600, or 14400
Parity	None
Data bits	8
Stop bits	1

- Various user-group bulletin boards (such as the bulletin-board services on the Association of PC User Groups network)
- In /SOFTLIB/MSLFILES on the Internet servers FTP.MICROSOFT.COM and WWW.MICROSOFT.COM

People within the United States who do not have a modem can order the Access Packs on disks by calling Microsoft Sales Information Center at (800) 426-9400 (voice) or (800) 892-5234 (text telephone). In Canada, you can call (905) 568-3503 or (905) 568-9641 (text telephone).

Keyboard Layouts for Single-Handed Users

Microsoft distributes Dvorak keyboard layouts that make the most frequently typed characters on a keyboard more accessible to people who have difficulty using the standard "QWERTY" layout. There are three Dvorak layouts: one for two-handed users, one for people who type with their left hand only, and one for people who type with their right hand only. The left-handed or right-handed keyboard layouts can also be used by people who type with a single finger or a wand. You do not need to purchase any special equipment to use these features.

Microsoft Windows and Microsoft Windows NT already support the two-handed Dvorak layout, which can be useful for coping with or avoiding types of repetitive-motion injuries associated with typing. To get this layout use the Windows Control Panel; consult your on-line documentation for detailed instructions. The two layouts for people who type with one hand are distributed as Microsoft Application Note GA0650. This application note is also contained in file GA0650.EXE on most network services and on the Microsoft Download Service. For instructions on obtaining this application note, see the preceding section, "Access Packs for Microsoft Windows and Microsoft Windows NT."

Microsoft Documentation in Alternative Formats

People who have difficulty reading or handling printed documentation may obtain many Microsoft publications from Recording for the Blind, Inc. Recording for the Blind distributes these documents to registered, eligible members of their distribution service, either on audio cassettes or on floppy disks. The Recording for the Blind collection contains more than 80,000 titles, including Microsoft product documentation and books from Microsoft Press. You can contact Recording for the Blind at the following address or phone numbers for information on eligibility and availability of Microsoft product documentation and books from Microsoft Press:

Recording for the Blind, Inc.
20 Roszel Road
Princeton, NJ 08540

Phone: (609) 452-0606
Fax: (609) 987-8116

Third-Party Utilities to Enhance Accessibility

A wide variety of third-party hardware and software products are available to make personal computers easier to use for people with disabilities. Among the different types of products available for the MS-DOS, Microsoft Windows, and Microsoft Windows NT operating systems are:

- Programs that enlarge or alter the color of information on the screen for people with visual impairments.
- Programs that describe information on the screen in braille or synthesized speech for people who are blind or have difficulty reading.
- Hardware and software utilities that modify the behavior of the mouse and keyboard.
- Programs that enable users to "type" using a mouse or their voice.
- Word or phrase prediction software that allows one to type more quickly and with fewer keystrokes.
- Alternate input devices, such as single switch or puff-and-sip devices, for those who cannot use a mouse or a keyboard.

For more information on obtaining third-party utilities, see "Getting More Information for People with Disabilities," later in this section.

Customizing Windows or Windows NT

There are many ways you can adjust the appearance and behavior of Windows or Windows NT to suit varying vision and motor abilities without requiring any additional software or hardware. These include ways to adjust the appearance as well as the behavior of the mouse and keyboard. The specific methods available depend on which operating system you are using. Application notes are available describing the specific methods available for each operating system.

See the appropriate application note for information related to customizing your operating system for people with disabilities. For information on obtaining application notes, see "Access Packs for Microsoft Windows and Microsoft Windows NT," earlier in this section.

Operating system	Application note
Microsoft Windows 3.0	WW0786.TXT
Microsoft Windows 3.1	WW0787.TXT
Microsoft Windows for Workgroups 3.1	WG0788.TXT
Microsoft Windows NT 3.1 and 3.5	WN0789.EXE
Microsoft Windows 95	WN1062

Getting More Information for People with Disabilities

For more information on Microsoft products and services for people with disabilities, contact:

Microsoft Sales Information Center	Voice telephone:	(800) 426-9400
One Microsoft Way Redmond, WA	Text telephone:	(800) 892-5234
98052-6393	Fax:	(206) 635-6100

The Trace R&D Center at the University of Wisconsin–Madison produces a book and a compact disc that describe products that help people with disabilities use computers. The book, titled *Trace ResourceBook*, provides descriptions and photographs of about 2,000 products. The compact disc, titled *CO-NET CD*, provides a database of more than 18,000 products and other information for people with disabilities. It is issued twice a year. To obtain these directories, contact:

Trace R&D Center	Voice telephone:	(608) 263-2309
S-151 Waisman Center	Text telephone:	(608) 263-5408
1500 Highland Avenue	Fax:	(608) 262-8848
Madison, WI 53705-2280		

For general information and recommendations on how computers can help specific people, you should consult a trained evaluator who can best match your needs with the available solutions. An assistive technology program in your area will provide referrals to programs and services that are available to you. To locate the assistive technology program nearest you, you can contact:

National Information System Center
for Developmental Disabilities
Benson Building
University of South Carolina
Columbia, SC 29208

Voice/text telephone: (803) 777-4435
Fax: (803) 777-6058

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Contributors to *Tutorials*

Gail Brown, Editor

Richard Carlson, Index Editor

Ted Chiang, Writer

Frank Crockett, Writer

David Adam Edelstein, Art Director

Pat Fenn, Production

Cathy Fisher, Proofreader

Jocelyn Garner, Writer

Kerry Lehto, Editor

Sibyl Lundy, Proofreader

Robert Reynolds, Illustrator

Chuck Sphar, Writer

Laura Wall, Writer