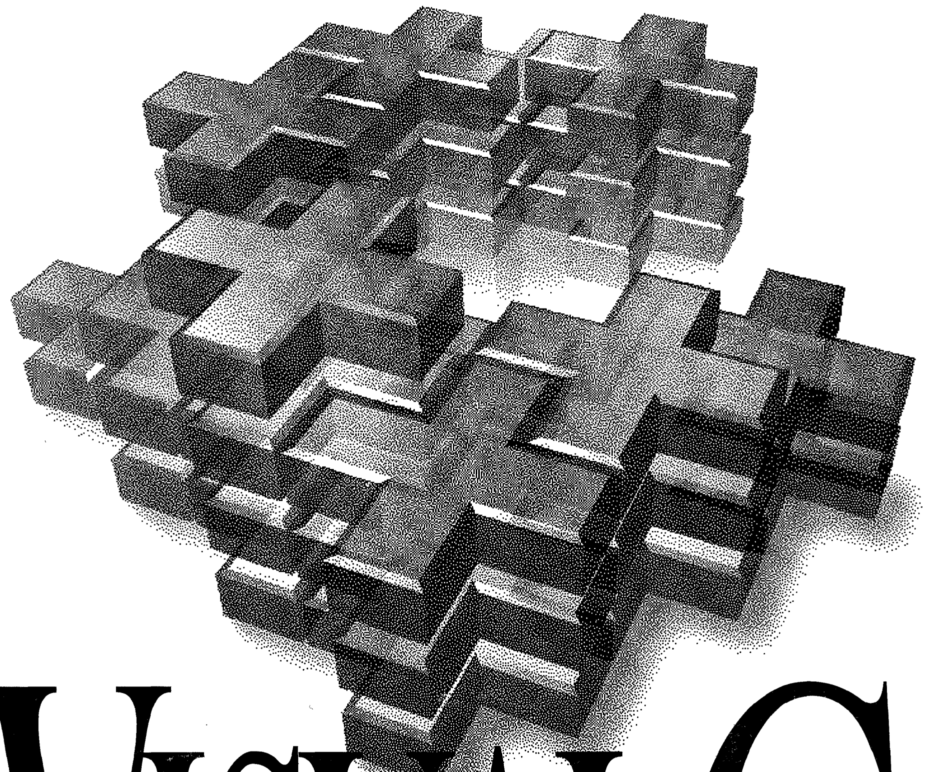
Introducing Visual C++



Microsoft® VISUAL C++™

Development System for Windows™ and Windows NT™

Introducing Visual C++

Microsoft® Visual C++™

Development System for Windows™ and Windows NT™
Version 2.0

Microsoft Corporation

Document No. DB57154-0694
Printed in the United States of America.

Contents

Figures and Tables

Tables

Introduction

The Microsoft® Visual C++™ 2.0 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 includes information to help you install and become familiar with Visual C++ and to help you understand how to use the main Visual C++ development tools. The book is divided into three parts:

- The Introduction provides installation information, tells how to use the online documentation, gives answers to common questions, and describes how to use the integrated environment to develop a Visual C++ application.
- The Tutorials demonstrate the use of various features of this product, including Object Linking and Embedding (OLE).
- The Appendix contains information on products that make the Microsoft Visual C++ development system more accessible for people with disabilities and on product support and services.

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, language-specific characters, 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.
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 double square brackets are optional.

Example	Description
#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 double 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 which 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.

PART 1

Getting Started

CHAPTER 1

Installing Microsoft Visual C++

This chapter describes how to install the Microsoft® Visual C++™ 2.0 development system for Windows™ and Windows NT™, as well as the Win32s™ files, from the CD-ROM.

Note The Visual C++ Cross-Development Edition for Macintosh is sold separately from this product. It is an add-on to Visual C++ 2.0, running on Windows NT 3.5, which enables Windows developers to use their existing Microsoft Foundation Class Library (MFC) and Win32 source code and tools expertise in porting applications from Windows to the Macintosh. In addition, the toolset includes a portability library that implements the Win32 API on the Macintosh with native Macintosh look and feel. This static linked library enables a single set of source code, written to either MFC or directly to the Win32 API, to target both Windows and the Macintosh. For more information contact the Microsoft Sales Fax Service at (800) 727-3351 (twenty-four hours per day, seven days per week).

System Requirements

Visual C++ requires the following minimum configuration:

- A PC with an 80386 or higher processor (80486 or higher recommended), running Microsoft Windows NT version 3.5.
- A VGA monitor (SVGA monitor recommended).
- 16 megabytes of available memory (20 megabytes recommended).
- A Windows NT-compatible CD-ROM drive connected to your computer.
- A hard disk with enough disk space to install the options you need. The setup program lets you select installation options and provides you with the disk space requirements for the options you select. It then checks to make sure you have enough space before copying files.

If you are developing and testing applications for Win32s, you also need either a dual-boot computer or a separate computer with:

- Microsoft Windows or Microsoft Windows for Workgroups version 3.1 running in enhanced mode.
- MS-DOS® version 5.0 or later.

You must install the Win32s dynamic-link libraries (DLLs) on the computer you plan to use for testing and debugging Win32s programs. For example, if you plan to use a second computer for testing and debugging Win32s programs, you must install the Win32s DLLs and debugger on the second computer. This requires that the second computer also have a CD-ROM drive.

Installing Visual C++

This section describes the basic installation for the Visual C++ development system and the Win32s files. The setup program provided by Visual C++ performs all tasks necessary for installing the Visual C++ components.

Quick Guide to Installation

Read this section as an overview to the installation procedures or simply for background information on the installation process.

The Setup Programs

There are two installation programs on the Visual C++ CD:

- \MSVC20\SETUP.EXE, the Visual C++ setup program
- \WIN32S\SETUP.EXE, the Win32s setup program

The Visual C++ setup program can be used to install Visual C++ to run from the CD-ROM drive or a network installation.

The Win32s setup program is used to install:

- Win32s system DLLs
- Win32s OLE DLLs
- Win32s tools
- Visual C++ remote debugging tools

Where to Start

If you are only developing applications targeted for Windows NT, run the Visual C++ setup program. This setup program must be run from Windows NT.

If you are developing applications for Win32s, run the Win32s setup program from Windows 3.1 to install Win32s on your target computer.

CD-ROM Directories

The Visual C++ CD-ROM root directory includes the directory \MSVC20 with the following subdirectories:

\MSVC20\BIN Executable files for applications and build tools for building 32-bit applications.

\MSVC20\INCLUDE C/C++ run-time and Microsoft Win32™ Software Development Kit (SDK) header files.

\MSVC20\LIB C/C++ run-time and Win32 SDK libraries.

\MSVC20\HELP Help files for Visual C++.

\MSVC20\MFC Subdirectories for all Microsoft Foundation Class Library files, including directories for libraries, source files, and include files.

\MSVC20\SAMPLES
Sample programs from the Microsoft Foundation Class Library, the Microsoft Win32 SDK, and Object Linking and Embedding (OLE).

\MSVC20\REDIST Files you may need to redistribute to users who receive your software.

\MSVC20\DEBUG Debug versions of files you may need to redistribute to users who receive your software.

Win32s files are located in the following \WIN32S subdirectories:

\WIN32S\BIN The Win32s debugging tool, remote debugging files, and Microsoft Profiler for Win32s, if you choose to install them.

\WIN32S\RETAIL Retail versions of the Win32s DLLs.

\WIN32S\DEBUG Debug versions of the Win32s DLLs.

\WIN32S\NLS Files associated with national language support (NLS).

\WIN32S\UT Files that comprise the universal thunker.

Installation Procedures

For help on any of the dialog boxes presented by the Setup program during installation, press the F1 key or choose the Help button in the dialog box.

You can install Visual C++ in one of the following ways:

Typical Installation This installs all tools and libraries for targeting applications for Windows NT. It also installs Microsoft Foundation Class Library files, sample files, and Help files.

Custom Installation This opens the Custom Installation dialog box. In this dialog box you can select files to install or files to add to your current installation. For more information, see "Custom Installation Options," on page 8.

Minimum Installation This installs the Visual C++ development environment, Microsoft Foundation Class Library files, and the tools and libraries needed for targeting applications for Windows NT.

CD-ROM Installation This copies the Visual C++ development environment from the CD-ROM drive to your hard disk and modifies Windows NT Registry keys for Visual C++ environment paths. When modified, the registry keys for executable files, include files, libraries, Help files, and Microsoft Foundation Class components point to the corresponding directories on the CD-ROM drive. For more information see "Installing Visual C++ to Run From a CD-ROM or Network Server" on page 12.

▶ **To run Setup**

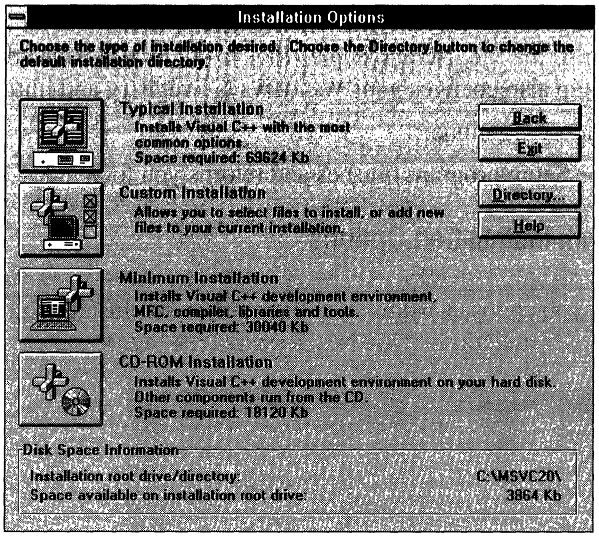
1. Place the Visual C++ CD in the CD-ROM drive.
2. Log on to Windows NT if you have not already done so.
3. Use the File Manager to run SETUP.EXE, located in the MSVC20 directory of the CD-ROM drive.

   –Or–

   From the File menu in Program Manager, choose Run and type `[X]:\MSVC20\SETUP` in the Command Line box (where `[X]` represents the drive letter associated with the CD-ROM drive).

   After initialization, Setup prompts you with a dialog box that describes the program and lets you continue or exit.
4. Choose Continue.

   The Installation Options dialog box appears (see Figure 1.1).



Installation Options

Choose the type of installation desired. Choose the Directory button to change the default installation directory.

Typical Installation
Installs Visual C++ with the most common options.
Space required: 69624 Kb

Custom Installation
Allows you to select files to install, or add new files to your current installation.

Minimum Installation
Installs Visual C++ development environment, MFC, compiler, libraries and tools.
Space required: 30040 Kb

CD-ROM Installation
Installs Visual C++ development environment on your hard disk. Other components run from the CD.
Space required: 18120 Kb

Disk Space Information
Installation root drive/directory: C:\MSVC20\
Space available on installation root drive: 3864 Kb

Figure 1.1 The Installation Options Dialog Box

5. If you decide to perform either the Typical Installation, Minimum Installation, or CD-ROM Installation, compare the disk space required for the option you have chosen (displayed next to the option) with the total disk space available (listed at the bottom of the dialog box).

   If you have enough disk space on the default drive (drive C), proceed to the next step.

   If you want to change the installation disk drive to one with more disk space, use the Change Directory dialog box (accessed from the Directory button) to change the drive.

6. To perform a default installation without customization, choose Typical Installation, Minimum Installation, or CD-ROM Installation.

   Or, if you want to customize the installation, choose Custom Installation.

7. If you chose Custom Installation in the previous step, select the options you want to install and choose Continue. For more information on custom installation, see the following section.

   Compare the amount of disk space required by the components you select with the total amount of disk space available. You may need to free some disk space before continuing, or install fewer components initially and add components later.

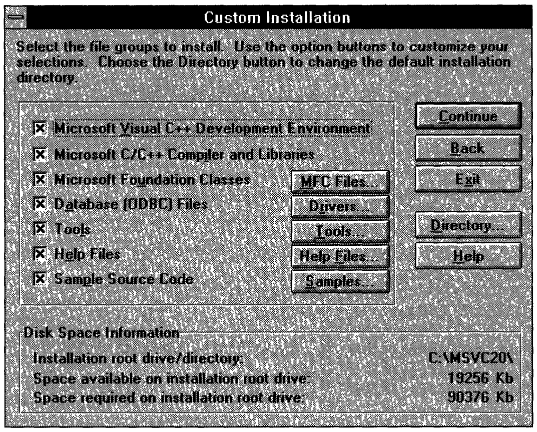
Setup checks for sufficient disk space, prompts you for registration confirmation, builds the file list, and copies files from the CD-ROM drive to your hard drive. Setup also changes your Windows NT registry and initializes ODBC.

8. Choose Continue.

   The Setup program finishes and returns you to Windows NT.

Custom Installation Options

When you choose Custom Installation from the Installation Options dialog box, the Custom Installation dialog box appears (see Figure 1.2).



Select the file groups to install. Use the option buttons to customize your selections. Choose the Directory button to change the default installation directory.

- [X] Microsoft Visual C++ Development Environment
- [X] Microsoft C/C++ Compiler and Libraries
- [X] Microsoft Foundation Classes
- [X] Database (ODBC) Files
- [X] Tools
- [X] Help Files
- [X] Sample Source Code

Disk Space Information

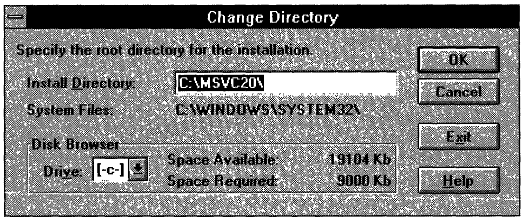
Installation root drive/directory:	C:\MSVC20\
Space available on installation root drive:	19256 Kb
Space required on installation root drive:	90376 Kb

Figure 1.2 The Custom Installation Dialog Box

This section describes each of the options available from this dialog box.

Directory

The Directory button opens the Change Directory dialog box, shown in Figure 1.3. You can change the default installation directory.



Specify the root directory for the installation.

Install Directory: C:\MSVC20\

System Files: C:\WINDOWS\SYSTEM32\

Disk Browser

Drive: [-c-]

Space Available: 19104 Kb

Space Required: 9000 Kb

Figure 1.3 The Change Directory Dialog Box

This dialog box is used to change to a disk drive with adequate space for the options you choose. You can also change the root directory for Visual C++ 2.0.

Microsoft Visual C++ Development Environment

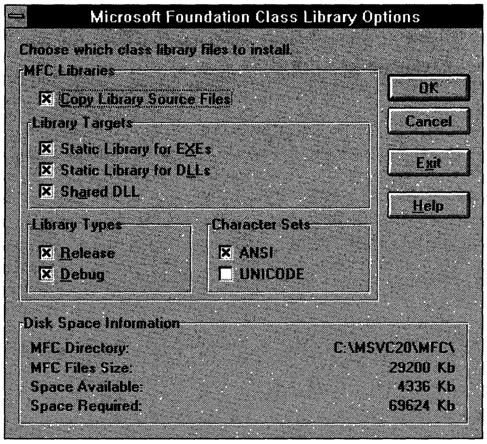
Select this check box to install the Microsoft Visual C++ development environment. This is installed in the \MSVC20\BIN directory.

Microsoft C/C++ Compiler and Libraries

Select this check box to install the Microsoft C/C++ 32-bit compiler and all 32-bit build utilities and libraries. The compiler and build utilities are installed in the \MSVC20\BIN directory. The libraries are installed in the \MSVC20\LIB directory. The include files are installed in the \MSVC20\INCLUDE directory.

Microsoft Foundation Classes

Select this check box to install the Microsoft Foundation Class files. These are installed in the \MSVC20\MFC directory and the Windows SYSTEM directory. You can use the MFC Files button to open the Microsoft Foundation Class Library Options dialog box (see Figure 1.4) and select which files to install.



Choose which class library files to install.

MFC Libraries
☒ Copy Library Source Files

Library Targets
☒ Static Library for EXEs
☒ Static Library for DLLs
☒ Shared DLL

Library Types
☒ Release
☒ Debug

Character Sets
☒ ANSI
☐ UNICODE

Disk Space Information
MFC Directory: C:\MSVC20\MFC\
MFC Files Size: 29200 Kb
Space Available: 4336 Kb
Space Required: 69624 Kb

Figure 1.4 The Microsoft Foundation Class Library Options Dialog Box

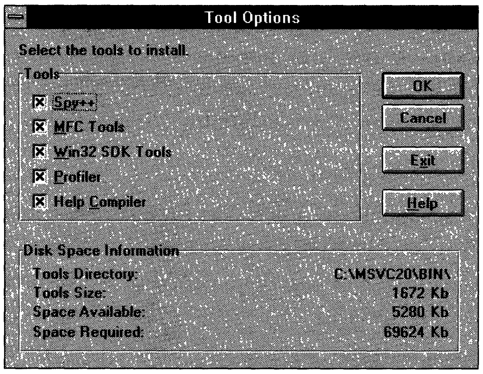
The Microsoft Foundation Class Library Options dialog box lets you install libraries for the following target types: static library for EXEs, static library for DLLs, and shared DLL. You can choose to install release and/or debug versions of these libraries, and to install the ANSI and/or UNICODE character sets. You can also choose to install the source files for the Microsoft Foundation Class Library.

Drivers

Select this button to open the ODBC Drivers dialog box, which you use to choose which ODBC drivers to install.

Tools

Select this check box to install Windows NT and OLE development tools. You can use the Tools button to open the Tool Options dialog box (see Figure 1.5) and select which tools to install. The tools are installed in the \MSVC20\BIN directory.



Select the tools to install.

Tools
- [x] Spy++
- [x] MFC Tools
- [x] Win32 SDK Tools
- [x] Profiler
- [x] Help Compiler

Disk Space Information

Tools Directory:	C:\MSVC20\BIN\
Tools Size:	1672 Kb
Space Available:	5280 Kb
Space Required:	69624 Kb

Figure 1.5 The Tool Options Dialog Box

The Tool Options dialog box has check boxes for six categories of tools. You can install the following tools:

Spy++
: A Windows NT-based utility that gives you a graphical view of the system's processes.

MFC Tools
: MAKEHM, which maps resource and command IDs in your application to Help contexts in Windows Help, and TRACER, which sets the value of **afxTraceEnabled** in order to enable or disable output from the **TRACE** macro.

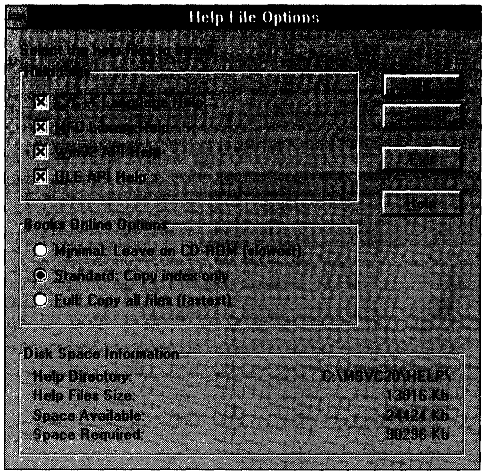
Win32 SDK tools
: Analysis tools such as WinDiff, PView, ZoomIn, and OLE SDK Tools.

Profiler
: A tool you can use to examine the run-time behavior of your programs.

Help Compiler
: The compiler and related tools used to create Windows Help.

Online Help Files

Select this check box to install Help files associated with Visual C++. You can use the Help Files button to open the Help File Options dialog box (see Figure 1.6) and select which files to install. Use the file size and disk space information in this dialog box to help determine which files to install. Help files are installed in the \MSVC20\HELP directory.



Select the help files to install

Help Files

[✓] C/C++ Language Help
[✓] MFC Library Help
[✓] Win32 API Help
[✓] OLE API Help

Books Online Options

○ Minimal: Leave on CD-ROM (slowest)
[✓ ● Standard: Copy index only]
○ Full: Copy all files (fastest)

Disk Space Information

```
Help Directory:        C:\MSVC20\HELP\
Help Files Size:              13816 Kb
Space Available:              24424 Kb
Space Required:               90296 Kb
```

Figure 1.6 The Help File Options Dialog Box

The Help File Options dialog box has check boxes so you can choose whether or not to install the following Help files:

C/C++ Language Help
: The language Help file, a reference to the C/C++ run-time library functions.

MFC Library Help
: The Help file containing Microsoft Foundation Class Library descriptions.

Win32 API Help
: The Help file for Win32 APIs.

OLE API Help
: The Help file for OLE APIs.

The Help File Options dialog box also offers the following choices for installing Books Online:

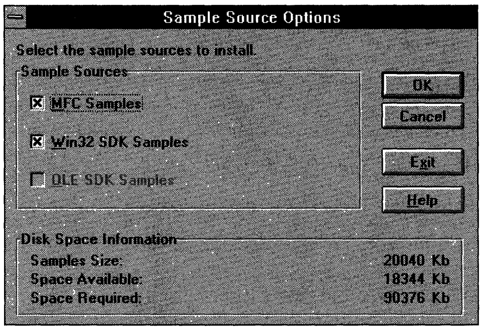
Minimal: Leave on CD-ROM
: Leaves all Books Online files on the CD-ROM. This option may affect how fast you can access Books Online.

Standard: Copy index only
: Installs on your hard disk the file that Books Online uses to find information. Content files remain on CD-ROM.

Full: Copy all files
: Installs all Books Online files on your hard disk. This option maximizes the speed at which you can access online information.

Sample Source Code

Select this check box to install sample source code. To modify the installation of sample code, choose the Samples button to open the Sample Source Options dialog box (see Figure 1.7).



Select the sample sources to install.

Sample Sources

[✓ MFC Samples]

[✓ Win32 SDK Samples]

☐ OLE SDK Samples

Disk Space Information

Samples Size:	20040 Kb
Space Available:	18344 Kb
Space Required:	90376 Kb

Figure 1.7 The Sample Source Options Dialog Box

The Sample Source Options dialog box lets you select from the following categories of sample files:

MFC Samples
: This includes Microsoft Foundation Class samples, which are installed in \MSVC20\SAMPLES\MFC.

Win32 Samples
: This includes all the Win32 SDK samples installed in the \MSVC20\SAMPLES\WIN32 directory.

OLE 2 SDK Samples
: This includes all the OLE 2 SDK samples. If you have the OLE 2 SDK and choose this option, Setup installs the OLE 2 SDK samples in the \MSVC20\SAMPLES\OLE2 directory.

Installing Visual C++ to Run From a CD-ROM or Network Server

If you have limited space on your hard drive, you may want to run Visual C++ directly from the CD-ROM drive. Visual C++ may run somewhat slower as a result of the increased access time for build utilities, libraries, include files and Help files. If performance becomes an issue, you can reinstall Visual C++ to run from the hard disk.

Warning If you are using a Write Once Read Many (WORM) drive, copy all of the files on the Visual C++ CD to the WORM drive. Run Setup from the WORM drive, using the CD-ROM installation option. This installs the Visual C++ development environment, AppWizard, and ClassWizard on your hard drive. All other Visual C++ components will run from the WORM drive. If you install Visual C++ on the WORM drive, as described in the following installation procedure, you will permanently lose some of your drive space every time you run Visual C++.

To install files on a network server so that they may be shared, copy the files onto the network server exactly as they appear on the CD-ROM. Then use the following procedure to install Visual C++ on individual network computers.

▶ **To install Visual C++ to run from the CD-ROM drive or network server**

1. Run Setup as described on page 6.
2. Choose CD-ROM Installation in step 6.

   The Setup program copies the Visual C++ development environment files onto your hard drive. It also modifies Windows NT Registry keys for Visual C++ environment paths to point to the directories on the CD-ROM or network server drive.

Installing Visual C++ to run from CD-ROM allows you to run the Visual C++ tools from the Visual C++ development environment or from the command line in a Windows NT command window. Note that you cannot build sample application projects that reside on the CD-ROM since it is a read-only medium. To build sample applications, copy the sample application's directory to your hard drive and build the application from there.

When Visual C++ is installed to run from the CD-ROM, it writes its default workspace and project files to the Windows directory.

Adding Libraries or Files to Your Installation

You can upgrade your Visual C++ installation by copying libraries or files you didn't install the first time. The following procedure describes how to do this.

▶ **To add libraries or files to your installation**

1. Run Setup as described on page 6.
2. Choose Custom Installation in step 6.
3. In the Custom Installation dialog box, clear all check boxes except the category you want to install.

4. If you have selected a category that has an associated dialog box, choose the associated button; otherwise skip to step 7.
5. Clear all default options and select the new option or options.

   For example, if you want to install Win32 API Help files, clear all options in the Custom Installation dialog box except Online Help Files, choose the Help Files button, and select only Win32 API Help in that dialog box.
6. Choose OK to close the associated category dialog box.
7. Choose Continue to proceed with the installation.

Installing Win32s

You can use Visual C++ to develop applications for Win32s; however, these applications must be debugged using a computer running MS-DOS and Windows 3.1. For this reason, the Win32s files are installed separately from the Visual C++ files.

You must install the Win32s files on a system that has a CD-ROM drive and runs MS-DOS, version 5.0 or later, and Windows 3.1 in enhanced mode. If you want to copy these files to another computer that doesn't have a CD-ROM, you should remember to modify the SYSTEM.INI and TOOLS.INI files on the second computer. See "Win32s Configuration" on page 16 for a description of what is changed in these files by the Win32s setup program.

Running Win32s Setup

The Win32s setup program copies files from the WIN32S subdirectories on the CD-ROM to your hard drive.

▶ **To run Win32s Setup**

1. Place the Visual C++ CD in the CD-ROM drive.
2. Run Windows 3.1 in enhanced mode.
3. Use the File Manager to run SETUP.EXE, located in the WIN32S directory of the CD-ROM drive.

   –Or–

   Choose Run from the Program Manager File menu and type `[X]:\WIN32S\SETUP` in the Command Line box (where `[X]` represents the drive letter associated with the CD-ROM drive).

   After initialization, Setup prompts you with a dialog box that describes the program and lets you continue or exit.

4. Choose Continue.

   The Installation Options dialog box appears. See the following section, "Win32s Installation Options," or choose the Help button for more information on options in this dialog box.

5. After selecting installation options, choose Continue.

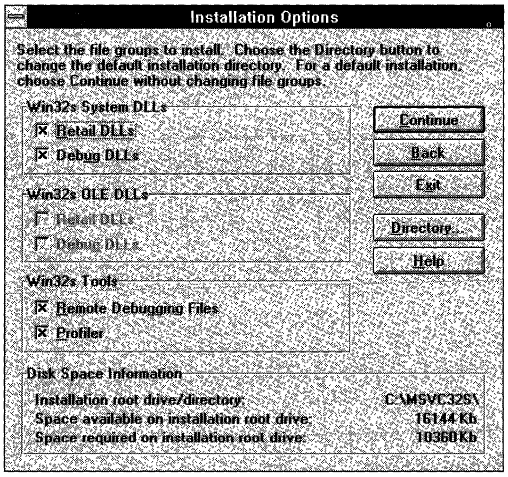
   Setup builds the file list, checks for sufficient disk space and copies files from the CD-ROM drive to your hard drive.

   After all files have been copied, a Configuration Files dialog box appears to let you choose to have Setup modify configuration files or write a modified version to another location.

6. Choose one of the options in the Configuration Files dialog box and choose Continue.

Win32s Installation Options

The Installation Options dialog box appears when you run the Setup program. (See Figure 1.8.)



Select the file groups to install. Choose the Directory button to change the default installation directory. For a default installation, choose Continue without changing file groups.

Win32s System DLLs
- [✓ Retail DLLs]
- [✓ Debug DLLs]

Win32s OLE DLLs
- Retail DLLs
- Debug DLLs

Win32s Tools
- [✓ Remote Debugging Files]
- [✓ Profiler]

Disk Space Information

Installation root drive/directory:	C:\MSVC32S\
Space available on installation root drive:	16144 Kb
Space required on installation root drive:	10360 Kb

Figure 1.8 The Installation Options Dialog Box

The Installation Options dialog box contains options for installing DLLs and Win32s tools. If you choose the Directory button, the Change Directory dialog box opens. Use the Change Directory dialog box to change the installation disk drive and directory.

Win32s DLLs

Retail and debug versions of Win32s system and OLE DLLs are available to install. At least one of the system DLLs must be installed on a system that runs applications that call Win32s APIs. The Win32s retail DLLs are first installed to the WIN32S\RETAIL directory. The Win32s debug DLLs are first installed to the WIN32S\DEBUG directory. If you choose to install debug files, some are copied to the Windows directory. If you choose to install the retail versions of the DLLs, or both retail and debug versions, the retail versions are copied to your Windows directory. You can run WIN32S\BIN\SWITCH.BAT to change from one version to the other.

Win32s Tools

If you select the Remote Debugging Files check box, the setup program installs files that enable you to debug programs on a remote system. If you select the Profiler check box, the setup program installs Profiler for Win32s.

Win32s Configuration

The Win32s setup program adds lines to the TOOLS.INI and SYSTEM.INI files when it installs Win32s files. See the Help file for more details. Modifications vary depending on which options you choose during installation. Remember to make these modifications if you copy the Win32s files to another computer.

Note If your CD-ROM is not supported by Windows NT, open Windows and refer to the Installation Notes in README.WRI.

CHAPTER 2

Using Online Documentation

Visual C++ online documentation consists of Quick Reference and Books Online. Quick Reference provides quick look-up information while you program. Books Online is the documentation set for Visual C++ in online format. Every Quick Reference topic has a link to Books Online, where complete information is available. The relationship between Quick Reference and Books Online is similar to that between a dictionary and an encyclopedia.

Visual C++ sets up Quick Reference files, by default, on your hard disk; Books Online files reside on your product CD-ROM. You can customize where you set up files or where you get information—you can even choose to go directly to Books Online for context-sensitive (F1) help.

This chapter describes how to use online documentation. Among the topics covered in this chapter are the following:

- Where to Find Information
- Browsing Topics
- Searching for Information
- Getting F1 Help on a Keyword or an Error Message
- Getting Help on a Dialog Box or Menu Command
- Getting Help on How to Accomplish a Task
- Customizing Quick Reference and Books Online
- Online Documentation Reference

Where to Find Information

Since Quick Reference is a subset of Books Online, some information that will be of interest to you will not be in Quick Reference. For instance, extended sample programs for the Microsoft Foundation Classes are not available in Quick Reference. However, Foundation Class sample programs are part of Books Online. The Visual C++ online documentation system also includes stand-alone help files that document various tools or libraries that support Visual C++.

Contents of Books Online

Books Online contains user's guides, programming guides, references, and reference sets, such as the OLE 2.0 SDK. Books are grouped so you can browse the material according to your interest.

Book Set	Books	Description
User's Guides	Introducing Visual C++	Describes how to install Visual C++ , introduces the product, and provides tutorials on using Visual C++ and the MFC library
	Visual C++ User's Guide	Describes the features of Visual C++ and all its tools
	Help Compiler User's Guide	Describes how to create application help files
MFC	Programming with MFC	Provides procedural and conceptual information about the MFC library
	Class Library Reference	Provides a complete description of the MFC library
	MFC Samples	Provides MFC library samples
	MFC Technical Notes	Provides technical notes written by and for programmers.
C/C++	Programming Techniques	Introduces programming techniques for new features and discusses migrating from other platforms and compilers
	C Language Reference	Describes the Microsoft implementation of C
	C++ Language Reference	Describes the Microsoft implementation of C++

Book Set	Books	Description
C/C++ (*continued*)	Run-Time Library Reference	Describes the Visual C++ run-time library
	iostream Class Library Reference	Describes the iostream class library
	Preprocessor Reference	Describes the C/C++ preprocessor
	Samples	Provides C and C++ samples
Win32 SDK	Win32 SDK, Volumes 1 through 5 and OpenGL	Describes the Software Development Kit for Win32
OLE 2.0 SDK	OLE 2 SDK	Describes the Software Development Kit for OLE 2
ODBC 2.0 SDK	ODBC SDK, Volumes 1 and 2	Describes the Software Development Kit for ODBC (Intel only)
Extensions: 68K Porting	Getting Started	Covers setup and installation of Visual C++ Cross-Development Edition for Macintosh
	Programmer's Guide	Includes overview, porting procedures, and reference information for the tools for the Cross-Development Edition.

Contents of Quick Reference

Quick Reference provides quick-reference information while you program. For example, by pressing F1 on a keyword or function in a source file, you open a Quick Reference topic that gives essential information about the keyword or function. These topics may include the following information:

- Prototype or syntax
- Compatibility, if applicable
- Return value
- Parameter descriptions
- A jump to a group of related functions or parent class
- An example opened from an Example button

Quick Reference also provides how-to information and context-sensitive help on dialog boxes and menu commands.

For information on	Choose the following Quick Reference file
Using online documentation	Using Online Documentation
Visual C++ tools	Visual C++
Module-definition file statements	Miscellaneous Tools
Resource-file statements	Miscellaneous Tools
Build errors	Build Errors
C/C++ language	C/C++ Language
Microsoft Foundation Classes	Foundation Classes
C/C++ run-time library	Run-Time Routines
iostream class library	iostream Classes
OLE 2.0 classes	Foundation Classes
Database classes	Foundation Classes
Win32 API	Windows API
OLE 2.0 API	OLE API
ODBC API	ODBC API (Intel only)
68K porting issues	68K Porting Reference (Intel only)

Note If you cannot find the information you need in a Quick Reference file, choose the Books Online icon in each Quick Reference topic or choose Search Plus in Books Online.

Contents of Stand-Alone Help Files

The table below lists stand-alone Visual C++ help files, which you cannot open from the Contents window. These files are either installed in the Visual C++ program group or in the product help directory.

For information on	Open the following help file
Visual C++ setup	Setup Help
Spy++	Spy++ Help
Windows NT Knowledge Base articles	Visual C++ 2.0 Knowledge Base
Technical support information	Technical Support Help
Windows Sockets specification	Windows Sockets
Windows NT sockets	Windows Sockets for NT
Windows NT Unipad editor help	Notepad Help
Hotspot editor for WinHelp files	Using Hotspot Editor

Browsing Topics

When Quick Reference or Books Online is opened, the content—represented by icons—is displayed in the Contents window. You can expand icons down a branch until you reach a topic. When you choose a topic in the Contents window, the Viewer window displays it. This side-by-side approach allows you to navigate without losing your place in the Contents window.

An icon represents a book or a classification that you can expand to a topic level. Clicking an icon always either expands or collapses the levels below it, in the same manner as File Manager. You can expand or collapse icons without opening the Viewer window. The Viewer window opens when you double-click a topic at the end of a branch.

Opening the Contents Window

The Contents window displays book sets or Quick Reference files. You expand a book set or Quick Reference file to display individual topics.

▶ **To open the Contents window**

- From the Help menu, choose Quick Reference or Books Online.

  –Or–

- From the Visual C++ program group, double-click the Quick Reference or the Books Online icon.

  –Or–

- From the Viewer window for Quick Reference or Books Online, choose the Contents button.

Once the Contents window is open, you can expand icons and jump to topics. You can also use the drop-down list in the Contents window to switch from Quick Reference to Books Online, or vice versa.

Navigating within the Contents Window

When the Contents window is opened, it displays a list of icons that represent categories of information at the same level.

In Quick Reference, for example, the list of icons at the top level represent individual files—for example, Foundation Classes Quick Reference or Windows API Quick Reference. In Books Online, a book icon can represent a book set, a single book, or an important subdivision within a book, such as a chapter.

Any icon can expand or collapse up or down a branch. For instance, when a book icon is expanded, the closed book icon is replaced with an open book icon.

▶ **To open or close an icon**

- Double-click the icon.

  The icon expands or collapses.

  Expand shortcut key: RIGHT ARROW

  Collapse shortcut key: LEFT ARROW

Note The color of an icon changes to indicate the level of information within a branch. There is no significance in color of an icon except to indicate its relative position in the branch.

▶ **To view a topic**

1. Double-click an icon. If another icon is displayed below it, double-click it until a topic icon appears.

   Shortcut key: ENTER

2. Double-click the topic icon.

   The Viewer window displays the topic.

   Shortcut key: ENTER

Navigating within the Viewer Window

Once you have opened a topic in the Viewer window, you can navigate to other topics in a number of ways.

▶ **To navigate within the Viewer window**

- Double-click a new topic in the Contents window

  The Contents window is the fastest way to browse or scan topics, since you decide which books to view and in which order.

- Choose the browse buttons on the Viewer window button bar

  The Browse buttons move you through topics one by one, forward or back. Since related material is developed in sequential topics, navigating with Browse buttons is a convenient way to view all topics on a particular subject.

- Choose a jump within a topic

  Jumps take you to topics that have a specific bearing on the current topic. These jumps may take you to a topic within a book or outside it.

- Choose Search on the Viewer window button bar

  All Quick Reference topics are indexed by keyword. Search allows you to look up keywords and display topics that contain the keywords. Search works best if you know exactly what you are looking for.

- Use Search Plus in Books Online to find information across all topics

  Search Plus allows you to jump to specific information within topics that span the entire Books Online library. Search Plus is the fastest way to find specific information.

You may open a topic in Quick Reference or Books Online and not know where you are. For instance, you may have entered a topic:

- After using Search
- After using Search Plus
- After jumping from another topic
- After browsing forward or back through several topics

▶ **To find out where you are**

- Choose the Contents button on the Viewer window button bar.

  If the Contents window is open, the icon containing the topic in the branch will expand and the correct topic will be highlighted. If the Contents window is closed, the Contents window will open and the same action will take place.

Searching for Information

You can search Visual C++ documentation in three ways:

- Help search
- Keyword Search
- Search Plus

Help search involves the Windows Help Search dialog box. Help search is confined to the current file in Books Online or Quick Reference. For instance, if you want to search for a Windows API but have the Run-Time Routines Quick Reference file open, your search will be empty.

Keyword Search is a search that is specific to Visual C++. The Keyword Search allows you to search for a language keyword, function, or build error across all Help files checked in the Help tab in the Options dialog box.

Search Plus is a full-text search. To take advantage of Search Plus, you must open Books Online—Quick Reference does not support Search Plus. You can limit your search by selecting a category or categories that correspond to the book sets in Books Online.

Help Search

You can quickly find information by using the Search button in the Viewer window. The Search button opens the Search dialog box, where you select a word that you want to search for. All topics associated with that word are listed.

▶ **To get help using Search**

1. In the Help button bar, choose the Search button.

   –Or–

   Type S.
2. Select the word or phrase you want to search for. As you type, the words that most closely match the text are displayed.
3. Choose the Show Topics button.
4. Select the topic you want to view. If necessary, use the scroll bar to see more topics.
5. Choose the Go To button.

Note If you open a Quick Reference file from the Help menu, the Search dialog box is automatically displayed.

Keyword Search

The Keyword Search command on the Help menu opens the Keyword Search dialog box. You can use the Keyword Search dialog box as a shortcut for opening help on a language keyword, library routine, class function, or error message.

You can determine which help files are searched. The default is all Quick Reference files.

Note You *cannot* get help on Visual C++ development environment from the Keyword Search dialog box. Keyword Search is restricted to language elements and build error message numbers.

▶ **To get help using Keyword Search**

1. From the Help menu, choose the Keyword Search command. The Keyword Search dialog box opens.
2. In the Keyword box, type a keyword, for example—CreateWindow, Open, or C1016.

   If the keyword appears in one or more files, the View Reference In box displays a list of the files in which it appears. You can then select which file you want. If the selected file has more than one reference within it, the file opens and displays the Help Search dialog box with the term selected.
3. Choose OK.

▶ **To determine which help files are searched with Keyword Search**

1. From the Tools menu, choose Options.

   The Options dialog box appears.
2. Select the Help tab.
3. Clear the files you don't want to search.

   You can check both Quick Reference or Books Online files. If you check both Foundation Classes Quick Reference and Foundation Classes in Books Online, the Keyword Search dialog box displays both files in the View Reference In list box when a keyword is entered.

Tip You may want to always search in Books Online rather than Quick Reference for certain APIs. The Help tab allows you to set your preferences. For instance, you may want to search the Windows API in Books Online but the Run-Time Routines in Quick Reference. The default setting is Quick Reference files.

The files you choose also sets which files open with context-sensitive help (F1 help from a source file or the output window). For more information, see "Customizing Context-Sensitive Help" on page 35.

Using Search Plus

Search Plus lets you search all or part of the books in Books Online to find a word or phrase. You can search for a single word, a combination of words in a topic, one word near another word, an exact phrase, and so on. You can narrow your search by defining a search pattern with Boolean expressions.

Search Plus displays the matches to your search in a window, from which you can select a topic to view. For example, to find all topics that have information on exceptions, you could search for "exceptions" or "exception handling."

The number of topics found by Search Plus is determined by both your search pattern and where you look for information.

▶ **To get help using Search Plus in Books Online**

1. Open the Contents window for Books Online.

   –Or–

   Open Books Online to a topic in the Viewer window.
2. Choose Search Plus on the Contents window toolbar.

   –Or–

   Choose Search Plus on the Viewer window button bar.
3. In the Find dialog box, type the word or words that you want to find.

   If you want to enter groups of words or exclude words, choose the Hints button. This displays the Search Hints dialog box. It describes the use of keywords to combine words, the use of quoting and grouping operators, and the use of a wild card for expanding a root word.

   You can also set the range used for the keyword NEAR. It determines how many other words can separate two words for which you are searching.
4. Under Look At, select either the Topic Titles Only or the All Text option button.
5. Under Look In, select the book set(s) in which you want to search. Use the scroll bar to see all book sets.
6. Choose the Search button.

   The titles of topics found are displayed in the Topics Found list. Topics in the current file have no label before the topic title. Topics in other files in the group are preceded by a label. These labels change as you switch from one file to another. See Table 2.1 for a listing of Book Sets and their corresponding labels.
7. Double-click the desired topic in the Topics Found list. Each entry in the topic displayed is highlighted.

   You can also use the Previous, Next, and Go To buttons to display topics.

Table 2.1 Book Set Help File Names

Book Set	Help File Name
User's Guides	VC20BKS1.HLP
MFC	VC20BKS2.HLP
C/C++	VC20BKS3.HLP
Win32 SDK	VC20BKS4.HLP
OLE 2.0 SDK	VC20BKS5.HLP
ODBC 2.0 SDK	VC20BKS6.HLP
Extensions	VC20BKS7.HLP

Narrowing Your Search

Sinçe a full-text search is comprehensive, you may find that the number of topics found is too large for practical purposes. For instance, searching for the word "dialog" would generate a large number of topics.

▶ **To narrow a search with Search Plus**

- Exclude files in the Look In group from the search list by clearing the check box for a book set.

  For example, searching for "dialog" in the User's Guides book set will give you an entirely different set of found topics than searching in the MFC book set. You can select or clear as many book sets as needed.
- Use the operators (AND, OR, and NOT) and the grouping operators listed in the Search Hints dialog box to create likely combinations of words.
- Use the " " quoting operators to search for an exact phrase.
- Use the NEAR operator with combinations of words to find only those that are close together in a topic.

Using Search Operators

The operators AND, OR, and NOT operate only on a topic level. The following Search For string finds any topic that contains both words:

```
menu AND command
```

The order of words in the Search For string does not matter. The following Search For strings produce identical results:

```
menu AND command
command AND menu
```

- The search mechanism ignores letter case. The same results appear if you search for cobject or COBJECT.
- You cannot search for the Search operators, such as NOT, for instance.

Built-in Exclusions from Searches

There are several built-in exclusions from searches. Most of these exclusions will have no effect on your searches, but you should be aware of them. For instance:

- You can search for alphanumeric characters only. Characters such as ~ [tilde], # [number sign], \ [backslash] and ([left parenthesis] are excluded from the search.
- Some words are excluded from the full-text search capability in order to build a smaller search index file. The excluded words, such as "because," are not usually helpful for finding information in this kind of material.

  This exclusion has a side effect with the NEAR keyword, because excluded words are essentially invisible to the search mechanism, even though they are visible to you on the topic page. For instance, if you set NEAR to 1 and search for:

  ```
  menus NEAR commands
  ```

  the search would find "commands because the menus" in the sentence "It may not display all the listed commands because the menus are dynamic," since both "because" and "the" are excluded from the search index.

Getting F1 Help on a Keyword or an Error Message

You can get help in a source or output window for keywords by placing the insertion point on a keyword or build error, then pressing F1. (A keyword for Help is any language element you want help on—a function, language keyword, message, class name, and so on.) You can get context-sensitive help by pressing:

If additional information on the keyword exists in Books Online, the Books Online icon in the Quick Reference topic is visible at the end of the topic. If an example exists for the keyword, the Example button in the button bar is active.

You can also open the Contents window to browse the contents of all Quick Reference files. For a quick look-up, try the Keyword Search command on the Help menu. For more information, see "Keyword Search" on page 28.

Pressing F1 in a Source or Output Window

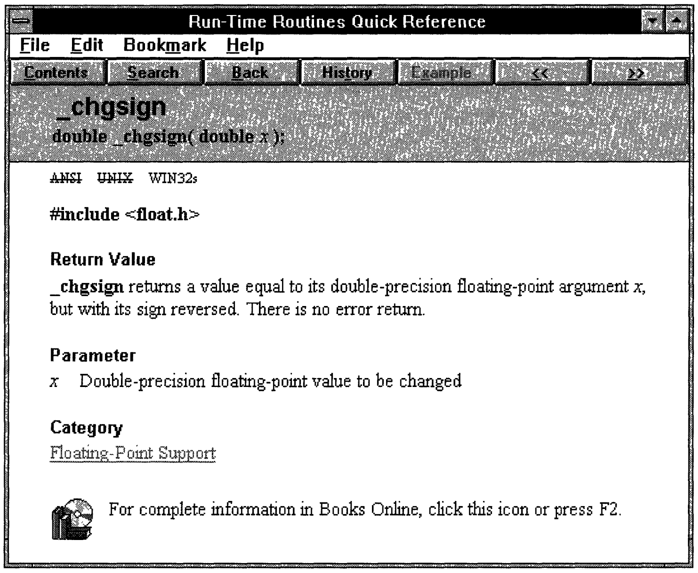
Pressing F1 on a keyword in a source or output window opens a topic in Quick Reference. When the same keyword exists in more than one file, the Keyword Search dialog box appears, listing the files in which the keyword help exists.

You can also press CTRL+F1 on a keyword to get help. CTRL+F1 search controls the order in which help files are searched, with the first topic found in any help file displayed. The help search stops once a topic is found. See “Customizing Context-Sensitive Help” on page 35 for information on determining a search order for context-sensitive help.

▶ **To get F1 help on a keyword or build error**

1. Place the insertion point anywhere on the keyword in a source window or anywhere on the error displayed in the output window.
2. Press F1.

   Help (Figure 2.1) on the keyword or error appears.


Run-Time Routines Quick Reference
File Edit Bookmark Help
Contents Search Back History Example << >>

_chgsign

double _chgsign(double *x*);

~~ANSI~~ ~~UNIX~~ WIN32s

#include <float.h>

Return Value

_chgsign returns a value equal to its double-precision floating-point argument *x*, but with its sign reversed. There is no error return.

Parameter

x Double-precision floating-point value to be changed

Category

Floating-Point Support

For complete information in Books Online, click this icon or press F2.

Figure 2.1 Typical Quick Reference Help Topic

3. To display an example in a second topic window, choose the active Example button.

   The Example help window appears.

4. To jump to Books Online for additional information, choose the Books Online icon.

You can choose which files Visual C++ looks in for help on a keyword. The default is all Quick Reference files. You can also set an option so that Visual C++ opens a help topic in Books Online rather than Quick Reference. See "Customizing Context-Sensitive Help" on page 35 for more information.

Note Clicking the right mouse button in a source or output window does not open Quick Reference. You may be familiar with this means of getting Quick Reference from earlier versions of QuickHelp. WinHelp does not support the right mouse button.

However, clicking the Right mouse button while a Quick Reference file is active does open a pop-up menu that lists the Quick Reference files. You can jump to another Quick Reference file by choosing a file on the menu.

Choosing Between Alternative Topics

Sometimes when you press F1 on a keyword in a source file, the Keyword Search dialog box appears. This occurs when help on the keyword exists in more than one Quick Reference file.

For example, help on the CreateBitmap function exists in both Foundation Classes Quick Reference and the Windows API Quick Reference, since both libraries include a CreateBitmap function.

If you request help on CreateBitmap from a source file, the Keyword Search dialog box lets you select either the topic in Foundation Classes Quick Reference or the topic in the Windows API Quick Reference.

▶ **To choose between alternative topics with F1 help**

1. Press F1 on a keyword in a source file.

   If alternative topics exist for the keyword, the Keyword Search dialog box (Figure 2.2) appears.

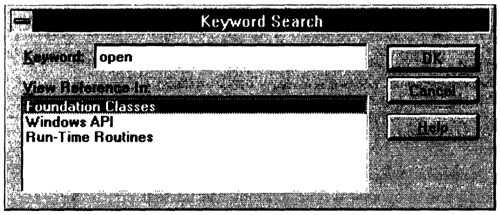


Figure 2.2 Keyword Search Dialog Box

2. Select the appropriate Quick Reference file and choose OK.

   If the selected Quick Reference file contains more than one instance of the help keyword, the Help Search dialog box for that help file opens after you choose a specific Quick Reference file.

3. From the Help Search dialog box, double-click the appropriate topic, then choose Go To.

   The Quick Reference topic appears in the Viewer window.

Getting Help on a Dialog Box or Menu Command

You can press F1 for context-sensitive Quick Reference on menu commands and dialog boxes.

Quick Reference on menus describes the purpose of each command. Quick Reference on dialog boxes describes the purpose of the dialog box and provides specific information for each control in the dialog box, such as list boxes, check boxes, text boxes, and option buttons.

▶ **To get help on a menu command**

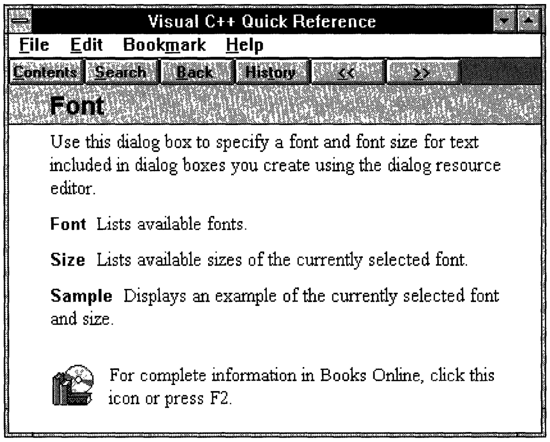
1. Open the menu.
2. Use the arrow key to highlight the command.
3. Press F1.

   Help on the menu appears.

▶ **To get help on a dialog box**

1. Open the dialog box.
2. Press F1 or choose Help.

Dialog box help (Figure 2.3) appears.


Visual C++ Quick Reference
File Edit Bookmark Help
Contents Search Back History << >>
Font

Use this dialog box to specify a font and font size for text included in dialog boxes you create using the dialog resource editor.

Font Lists available fonts.

Size Lists available sizes of the currently selected font.

Sample Displays an example of the currently selected font and size.

For complete information in Books Online, click this icon or press F2.

Figure 2.3 Typical Dialog Box Help Topic

If any related procedures exist in Quick Reference, they are listed. The Books Online icon jumps to a topic in Books Online where the function of the dialog is generally discussed.

Getting Help on How to Accomplish a Task

Visual C++ Quick Reference includes information on tools that make up Visual C++: the text editor, the resource editors, the debugger, and so forth. Visual C++ Quick Reference is divided into two parts, Step-by-step Instructions and Reference Information.

The Step-by-step Instructions part contains brief topics that describe how to accomplish a task, such as setting a breakpoint. To get background information or greater detail, choose the Books Online icon at the bottom of the topic. The Reference Information part describes all features of the interface such as dialog boxes, toolbars, and windows.

Note Quick Reference for the Intel version of Visual C++ includes information on the Visual C++ 2.0 Cross-Development Edition for Macintosh. Step-by-step Instructions and Reference Information parts for the Cross-Development edition are listed under the 68K Specific node.

▶ **To open Visual C++ Quick Reference**

- From the Help menu, choose Quick Reference.

  –Or–

- From the Visual C++ program group, choose Quick Reference.

  –Or–

- From the Books drop-down list in the Contents window, choose Quick Reference.

Getting Help on How-To Topics

The how-to topics in Visual C++ Quick Reference are organized by tools.

▶ **To get help on a how-to procedure**

1. Open Visual C++ Quick Reference.

   The Contents window opens, displaying Quick Reference files.

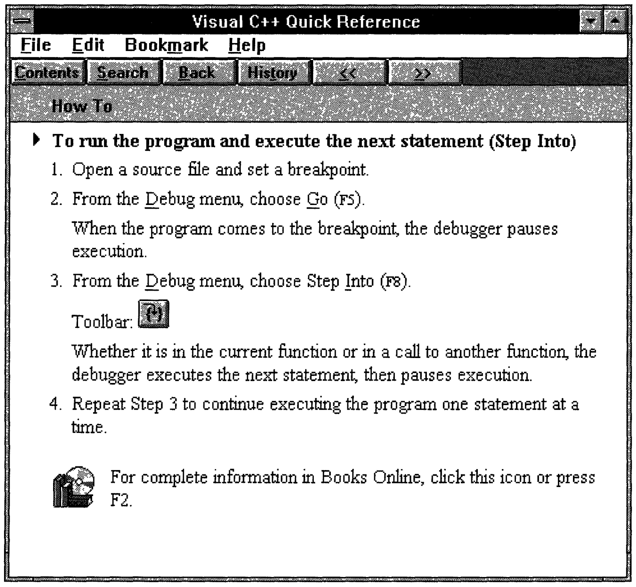
2. Double-click Visual C++.
2. Double-click Step-by-step Instructions.

   The categories of how-to topics are displayed.

3. Double-click Debugger, for example.
4. From the list of topics, double-click:

```
To run the program and execute to the next statement
```

The how-to topic (Figure 2.4) appears in the Viewer window. See Figure 2.4 below.


Visual C++ Quick Reference
File Edit Bookmark Help
Contents Search Back History << >>
How To

▸ **To run the program and execute the next statement (Step Into)**

1. Open a source file and set a breakpoint.
2. From the Debug menu, choose Go (F5).

   When the program comes to the breakpoint, the debugger pauses execution.

3. From the Debug menu, choose Step Into (F8).

   Toolbar:

   Whether it is in the current function or in a call to another function, the debugger executes the next statement, then pauses execution.

4. Repeat Step 3 to continue executing the program one statement at a time.

For complete information in Books Online, click this icon or press F2.

Figure 2.4 Typical How-To Help Topic

For additional information on this or any other topic, choose the Books Online icon at the end of the topic.

Note You can also open Search from the Viewer window button bar to find how-to procedures in Visual C++ Quick Reference.

Customizing Quick Reference and Books Online

Visual C++ allows you to copy the Books Online files to your hard disk or a network location. You also have the flexibility to determine where Visual C++ looks for help—in Quick Reference, Books Online, or a combination of the two.

Installing Books Online to Your Hard Disk or to a Network Location

Installing Books Online on your hard disk or a network location requires changing path information in the Registry. You can do this by hand, but the Visual C++ Setup program will do it for you.

▶ **To install Books Online to another location**

1. Run Setup from the Visual C++ installation disk.

   The Setup program opens.
2. In the Welcome dialog box, choose Continue.

   The Installation Options dialog box appears.
3. Choose the Directory button.

   The Change Directory dialog box appears.
4. Enter the location where you want to copy the files. Choose OK.

   The Installation Options dialog box appears.
5. Choose Custom Installation.

   The Custom Installation dialog box appears.
6. Clear the check boxes for all options but Help Files.
7. Choose Help Files.

   The Help File Options dialog box appears.
8. Clear the check boxes for all help files.
9. In the Books Online Options group, select the Full: Copy All Files option.

Customizing Context-Sensitive Help

You can limit your F1 searches to a set of files or determine a search in a specific order through a set of files for CTRL+F1 help. The files you select can be part of Quick Reference or Books Online. The default is all Quick Reference files.

For example, suppose you want F1 help to open Quick Reference files except for the Windows API, in which case you wanted help to jump to Books Online. When you pressed F1 on a CreateWindow, for example, Visual C++ would list the files in which CreateWindow appears—but only in the files you have selected.

You can also determine a search order by moving files up or down a list. When you press CTRL+F1 on a keyword, Visual C++ looks through the files in order for the help topic on the keyword. Help displays the first topic found.

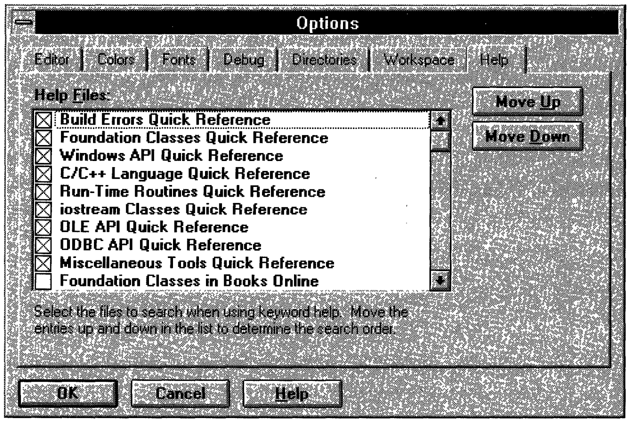
▶ **To limit the number of help files searched for F1 help**

1. From the Tools menu, choose Options.

   The Options dialog box appears.

2. Select the Help tab.

   The Help tab (Figure 2.5) appears.



☒ Build Errors Quick Reference
☒ Foundation Classes Quick Reference
☒ Windows API Quick Reference
☒ C/C++ Language Quick Reference
☒ Run-Time Routines Quick Reference
☒ iostream Classes Quick Reference
☒ OLE API Quick Reference
☒ ODBC API Quick Reference
☒ Miscellaneous Tools Quick Reference
☐ Foundation Classes in Books Online

Select the files to search when using keyword help. Move the entries up and down in the list to determine the search order.

Figure 2.5 Help Tab In the Options Dialog Box

3. Clear the check boxes of the Quick Reference files you don't want to search in the Help Files list box.

When you press F1 on a keyword, only the file or files you have selected are searched.

Note Both context-sensitive F1 help and Keyword Search look only in the files you have checked in the Help Files list box. If you have unchecked all files or unchecked the file in which the keyword help exists, pressing F1 in a source file or an output window will generate this error:

```
Help Topic Not Found
```

▶ **To customize an F1 search order**

1. From the Help tab on the Options dialog box, select a help file name in the Help Files list box.
2. Choose Move Up or Move Down.
3. Repeat Steps 1 and 2 until files are ordered as desired.

▶ **To search for a keyword in a customized search order**

1. Place the insertion point anywhere on the keyword in a source window.
2. Press CTRL+F1.

Visual C++ looks for keyword help in the Help Files list box order and displays the first topic found.

For example, suppose you have both Foundation Classes Quick Reference and Foundation Classes in Books Online checked in the Help Files list box. If you have moved Foundation Classes in Books Online to the top of the list, pressing CTRL+F1 on a member function would open the member function help topic in Books Online.

If you pressed F1 on the same member function, the Keyword Search dialog box would appear, listing both the Quick Reference and Books Online Foundation Class files. CTRL+F1 help gives you the flexibility to get the level of information you want first on a case-by-case basis.

Tip Use CTRL+F1 for the Macintosh cross-development porting information. Context-sensitive help will always open to the porting information first. You can then jump to the appropriate function or API in another file if you need more information.

Online Documentation Reference

This reference describes interface features, shortcut keys, and book-set filenames for Visual C++ online documentation.

Contents Window Controls

Control Menu

Font

Opens the Fonts dialog box, in which you can change the font and font size for Contents window display.

About Contents

Opens a dialog box that gives information about the Contents window.

Drop-down List Box

Allows you to select either Quick Reference or Books Online for display in the Contents browser.

Search Plus

For Books Online, opens the Find dialog box to begin a full-text search. Disabled in Quick Reference.

Viewer Window Buttons

Contents

If the Contents window is opened, highlights the title of the current topic in Quick Reference or Books Online. If the Contents window is closed, it opens the Contents window and highlights the current topic.

Search

Displays the words you can use to search for related topics in the current help file. Use this button to look for topics related to a word. Search is not a full-text search across the Books Online library, but works like an index for the current file.

Search Plus—*Books Online Only*

Opens the Find dialog box to begin a full-text search one or more book sets.

Back

Displays the topics you viewed, in reverse order.

History

Displays a list of the last 40 topics you viewed in the Windows Help session. The most recently viewed topic is listed first. To open a topic, double-click it.

Example

When active, opens a programming example in a second topic window.

Jumps to related—and usually additional—information in Books Online.

Shortcut Keys

Context-Sensitive Help

To	Press or Click
Open help topic in a source or output window	F1
Open first keyword help topic found in a list of files	CTRL+F1
Display pop-up menu listing Quick Reference files	Right mouse button while Quick Reference help file is in focus.

Contents Window Shortcut Keys

To	Press
Select the first book	HOME
Select the last book	END
Select the next level down	DOWN ARROW
Select the next level up	UP ARROW
Page down the list of books	PAGE DOWN
Page up the list of books	PAGE UP
Expand the current level	RIGHT ARROW
Collapse the current level	LEFT ARROW
Use shortcut for double-click	ENTER

Book Sets and Corresponding Help File Names

Book Set	Help File Name
User's Guides	VC20BKS1.HLP
MFC	VC20BKS2.HLP
C/C++	VC20BKS3.HLP
Win32 SDK	VC20BKS4.HLP
OLE 2.0 SDK	VC20BKS5.HLP
ODBC 2.0 SDK	VC20BKS6.HLP
Extensions	VC20BKS7.HLP

CHAPTER 3

Questions and Answers

The questions in this chapter represent a cross-section of typical questions asked by developers about the Visual C ++ development system and its components.

Questions and answers are provided on the following topics:

- Installation
- Cross-platforms
- General information
- Development environment
- Microsoft Foundation Class Library (MFC)

Installation

Q. What does the Setup program do when it modifies the Windows NT registry?

A. It registers the directories for the executable files, include files, library files, Books Online files, and Help files used by Visual C++. If you install Visual C++ to run from your hard drive, the drives and directory paths specified at installation are registered. If you install Visual C++ to run from the CD-ROM, the CD-ROM drive and directory paths are registered.

The setup program also modifies the settings for the Windows NT Automatic Debugger for Just-In-Time Debugging. Finally, to permit debugging of OLE servers and clients, the Setup program modifies OLE Remote Procedure Call settings.

Q. How do I change the default installation directory?

A. During installation, choose the Directory button in the Installation Options dialog box and type a new directory name in the Change Directory text box.

Q. Can I run Visual C++ from the CD-ROM?

A. Yes, an option in the Installation Options dialog box of the Setup program lets you run Visual C++ from the CD-ROM or from a network. This installation will install Visual C++ and its wizards—AppWizard and ClassWizard—on your hard drive. If you choose to run Visual C++ from the CD-ROM, you'll save space on your hard disk, but Visual C++ performance may be negatively affected.

Q. How do I uninstall Visual C++ 2.0?

A. You can uninstall Visual C++ 2.0 by deleting the root directory of Visual C++, several DLLs located in the Windows NT System32 directory, and an entry from the Windows NT registry.

▶ **To uninstall Visual C++ 2.0**

1. Open File Manager.
2. Select the MSVC20 directory.

   The directory is located on drive C by default. If you installed Visual C++ 2.0 at a location other than C:\MSVC20, select that directory.
3. Move out of the directory any files you want to save.
4. From the File menu, choose Delete.

   The directory and its files are deleted.
5. Choose the SYSTEM32 subdirectory of your Windows NT directory.
6. Select files in the SYSTEM32 directory that are listed in the REDIST.WRI file.
7. From the Edit menu, choose Delete.

   The files you selected are deleted from the directory.
8. Open Regedit32.
9. Open the Software folder.
10. Open the Microsoft folder.
11. Select the Visual C++ 2.0 folder.
12. From the File menu, choose Delete.

    The entries for Visual C++ 2.0 are deleted from the Windows NT registry.

Q. How do I uninstall Win32s?

A. You can uninstall Win32s by deleting the root directory of Win32s, several files located in the Windows system directory, and several entries from the SYSTEM.INI file.

▶ **To uninstall Win32s**

1. Open File Manager.
2. Select the WIN32S directory.

   The directory is located on drive C by default. If you installed Win32s at a location other than C:\WIN32S, select that directory.
3. From the File menu, choose Delete.

   The directory and its files are deleted.
4. Choose the system subdirectory of your Windows directory.
5. Select files in the System directory that are listed in the REDIST.WRI file.
6. From the File menu, choose Delete.

   The files you selected are deleted from the directory.
7. Open the System Configuration editor (SYSEDIT.EXE).
8. Make the SYSTEM.INI window active.
9. Delete the following entry from the [386Enh] section of your SYSTEM.INI file:

   ```
   device=C:\WINDOWS\SYSTEM\WIN32S\W32S.386
   ```

   where C:\WINDOWS\SYSTEM is the windows system directory.
10. Delete the following entry from the [Boot] section of your SYSTEM.INI file:

    ```
    drivers=WINMM16.DLL
    ```
11. From the File menu, choose Save to save changes to the SYSTEM.INI file.

Cross-Platforms

Q. How do I port my 16-bit application for Windows to Win32?

A. First, read the section titled "Porting 16-Bit Code to 32-Bit Windows" in *Programming Techniques* to learn how to port applications to Windows NT. Then use the Microsoft PortTool application, located in the \MSVC20\BIN directory, to help identify areas in your program that require revision. For more information, see the article "MFC: Porting MFC Applications to 32-Bit" in *Programming with the Microsoft Foundation Class Library.*

Q. How do I port my character-mode applications to Windows NT?

A. Character-mode applications that use standard I/O calls such as **printf** and **scanf** can be built in Windows NT as console applications. If your code uses BIOS calls, graphics libraries, or direct hardware calls, you can use that code for applications that run under Windows NT.

Q. Can I write or debug 16-bit applications using Visual C++ 2.0?

A. No. Visual C++ 1.5 is available for developing 16-bit applications that run in Windows 3.1 and MS-DOS.

Q. Can I debug Win32s programs using the debugger?

A. Since Win32s applications use a subset of the Win32 APIs, you can use the debugger to debug under Windows NT. When you are debugging only Win32s errors, you can use remote debugging.

Q. How do I use libraries to build a Win32s executable under Windows NT?

A. Link with the same libraries used for any Win32 application. This is enabled automatically when you choose Windows application (.EXE) as a project type.

Q. Can I port my Windows applications to run on Macintosh?

A. Yes, you can buy the Visual C++ 2.0 cross-platform edition as an add-on to Visual C++. The add-on product provides a set of NT-hosted tools for recompiling your Windows code for the Motorola® 680x0 processor and a portability library for implementing Windows on Macintosh. This enables you to develop GUI applications with a single source-code base (written to the Win32 API) and implement them on Microsoft Windows or Apple® Macintosh platforms.

Q. How do I port a Windows application to Macintosh?

A. The general steps required to port a Windows application to Macintosh are:

1. Port your application from Windows 16-bit code to 32-bit code.

   This might be the most time-consuming part of the job.

2. Segregate those parts of your application that are unique to Windows from those parts that are specific to the Macintosh.

   This may involve using conditional compilation, or it may require changing the source tree for your project.

3. Port your Win32 API code to the Macintosh using the portability library for the Macintosh. Use Visual C++ 2.0 to compile, link, and debug your code.
4. Write Macintosh-specific code to take advantage of unique Macintosh features.

Q. Is the Microsoft Foundation Class Library supported on the Macintosh?

A. Yes, MFC has been ported to the Macintosh. Writing to MFC provides you the greatest degree of portability. Because MFC provides a greater level of abstraction than writing directly to the Win32 API, it is easier to implement MFC correctly on the Macintosh.

General Information

Q. Does Visual C++ 2.0 include templates?

A. Yes. Visual C++ 2.0 handles templates, which enable you to define a family of functions or classes that can operate on different types of information.

Q. Does Visual C++ 2.0 support exception handling?

A. Yes. Visual C++ 2.0 handles anomalous situations, known as "exceptions," which may occur during the execution of your program.

Q. How do I create 32-bit OLE Custom Controls with Visual C++?

A. The OLE Control Developer's Kit is available as an add-on to Visual C++. This kit includes extensions to the Microsoft Foundation Class Library, Wizards, and other tools that enable you to create OLE Custom Controls.

Q. How do I port my Win32 Software Development (SDK) Projects to Visual C++?

A. You have two options, depending on how integrated you want your Win32 SDK projects to be with Visual C++:

- Update the CL options in the Win32 SDK makefiles and run them from the command line or as an external project. Visual C++ doesn't support the CL MIPS options and treats the /Zi options differently. The LINK options are fully compatible. See Chapter 2, "Working With Projects," Chapter 20, "Setting Compiler Options," Chapter 21, "Setting Linker Options," Appendix A, "CL Reference," and Appendix B, "LINK Reference" in the *Visual C++ User's Guide*.
- Re-create your projects as Visual C++ projects. See Chapter 2, "Working With Projects"; Chapter 20, "Setting Compiler Options"; and Chapter 21, "Setting Linker Options," in the *Visual C++ User's Guide* for information on creating Visual C++ projects and selecting compiler and linker options.

Q. Can I use Visual C++ to write POSIX or OS/2® applications?

A. No. The Visual C++ development system is designed for developing console or GUI applications for the Win32 subsystem. You can use the Win32 Software Development Kit to build POSIX applications.

Q. Does Visual C++ target MIPS?

A. No, only Intel®-compatible microprocessors are supported by Visual C++ 2.0. For developers who target MIPS, a VC++ 2.0 for MIPS edition is available.

Q. Do I need the Win32 SDK if I have VC++ 2.0?

A. No.

Q. Can I run all my applications and DLLs under Win32s?

A. You can run applications and DLLs under Win32s provided that they use features supported on Win32s. For example, Win32s does not support multithreading, so an application that includes multithreading would not run correctly on Win32s. For a list of APIs supported on Win32s, see the WIN32API.CSV file.

Development Environment

Q. Can I do mixed-language programming in Visual C++?

A. You can mix C and C++ source code; however, Visual C++ does not directly support mixing other languages such as assembly language with C or C++ except by using the **__asm** keyword to inline assembly instructions. You can integrate any language, such as Microsoft Macro Assembler (MASM), with the Visual C++ tools by installing the compiler on the Tools menu and adding the resulting object files to your C or C++ project. Alternatively, you can modify your C or C++ internal project makefile to include mixed-language build statements and then load the resulting makefile as an external project.

Q. What is Just-In-Time Debugging?

A. Just-In-Time debugging is a new feature of VC++ 2.0. It enables you to open a debugging session when an error occurs instead of closing your application. For example, if an unhandled exception occurs and Just-In-Time debugging is enabled, Visual C++ 2.0 uses a Win32 service to notify the VC++ debugger that an error has happened. The debugger starts and attaches to your application at the point of fault. This feature can save you time because you don't have to re-create errors.

The Just-In-Time debugger is also useful when you are debugging asserts in place. If an assertion occurs when Just-In-Time debugging is enabled, you can choose the Retry button on the assertion message to open the debugger at the point of the assertion.

▶ **To enable Just-In-Time debugging**

1. From the Tools menu, choose the Options command.
2. Select the Debug tab.
3. Select the Just-In-Time Debugging check box.
4. Choose OK.

Q. How do I display memory addresses in the integrated debugger?

A. Use the Memory window, opened from the Debug menu, to display memory addresses. To display the value of a memory address, place the insertion point in the memory address field and type the address you want to examine.

Q. How do I see which command-line options are passed to the compiler?

A. This information is shown in the Project Settings dialog box.

▶ **To view command-line options for a file**

1. From the Project menu, choose Settings.

   Visual C++ displays the Project Settings dialog box.
2. In the Settings For list, choose the target, targets, or source file whose settings you want to view.
3. Select one of the following tabs: C/C++, Link, Resources, OLE Types, or Browse Info.

   Visual C++ displays options associated with the category you selected.

Q. Can I use project files from the Visual C++ development system for Windows in the Visual C++ development system for Windows NT?

A. Yes, Visual C++ 2.0 converts 16-bit project files.

Q. How do I change the base of my variables in the Watch window from decimal to hexadecimal?

A. The quickest way to do this is to use the shortcut menu associated with the Watch window.

▶ **To switch to hexadecimal display in the Watch window**

1. Place the mouse pointer over the Watch window, and then press the right mouse button to display the shortcut menu.
2. Choose the Hexadecimal Display command from the shortcut menu.

Q. Can I edit 8- or 24-bit-per-pixel color bitmaps?

A. Yes.

Q. Can I use Visual C++ as a general-purpose resource editor?

A. Yes, Visual C++ is a powerful resource editor when used with traditional SDK applications or with MFC applications. You can use ClassWizard to create MFC code to support resource objects.

Microsoft Foundation Class Library

Q. Can I use VBX custom controls with Visual C++ applications?

A. VBX custom controls are 16-bit dynamic-link libraries and can only be used with 16-bit applications.

Q. What is the difference between the 16-bit and 32-bit versions of the Microsoft Foundation Class Library?

A. The message handlers, **CWnd::OnCommand** and **CWnd::OnParentNotify,** and the **CTime** class have changed. Also, the 32-bit version of MFC does not support VBX controls or the Microsoft Windows for Pen Computing extensions. For more information, see the article "MFC: Changes from MFC Versions 2.0 and 2.5" in *Programming with the Microsoft Foundation Class Library.*

Q. Can I use ClassWizard to recognize my own classes?

A. No. ClassWizard only recognizes classes that have been added using ClassWizard. These, by definition, are derived from the framework classes.

Q. Can I use AppWizard to create C or C++ code without Microsoft Foundation Class Library support?

A. No. AppWizard is specifically designed to create Microsoft Foundation Class Library version 3.0 application source code.

CHAPTER 4

Developing a Microsoft Visual C++ Application

This chapter describes how to use Visual C++ to develop an application and presents an overview of the Microsoft Foundation Class Library (MFC) and the development process.

We recommend that you complete the tutorials later in this guide to learn how to develop a Visual C++ application. You can read this chapter before beginning the tutorials or as a quick refresher.

What Is a Visual C++ Application?

A Visual C++ application is an application for Windows that you design and develop using MFC, the Microsoft Visual C++ build tools, and the Visual C++ Windows-hosted development environment.

By using a totally integrated environment, you can develop your application by focusing on the visual interface elements. Visual C++ calls these elements "user-interface objects." You first design the user-interface objects and then use Visual C++ tools to create and manage the code to support them. These tools automate the often tedious and error-prone process of deriving classes, creating member functions, and mapping functions to messages. Using Visual C++ tools, you can concentrate on designing the resources for your application and writing the functional code to handle messages.

You can also use the Visual C++ tools to develop standard applications for Windows SDK in C or C++, since Visual C++ includes a text editor, project window, browse window, debugger, and resource editors. If you are familiar with SDK programming for Windows, you'll find that the Visual C++ tools make the transition to object-oriented program development—and MFC—easier than you might have imagined.

What Is the Class Library?

The Microsoft Foundation Class Library (MFC) enables C++ programmers to write applications for Microsoft Windows. The class library gives you a complete "application framework" which defines an architecture for integrating the user interface of an application for Windows with the rest of the application. It also provides implementations for a large set of the user-interface components described in *The Windows Interface: An Application Design Guide*, available from Microsoft Press.

This section introduces the application framework and its associated visually-oriented programming tools: Visual C++, AppWizard, and ClassWizard. For more information about the class library, see Chapter 1 of *Programming with the Microsoft Foundation Class Library.*

The Framework

MFC is a group of C++ classes collectively known as an application framework. These classes provide the framework and essential components of an application for the Windows operating system. The purpose of the framework is to reduce the effort required to design and implement applications for Windows. The framework embodies the accumulated wisdom of experienced programmers for the Windows operating system.

Note In the documentation, you'll see the terms "application framework" and "framework" used interchangeably. The classes that make up the framework are listed and explained in the *Class Library Reference*. For an explanation of how the framework works, see Chapter 1 of *Programming with the Microsoft Foundation Class Library.*

The application framework supplied by MFC is powerful and easy to reuse because the framework is an object-oriented class library. Instead of editing the framework's source code directly, you derive new, specialized classes from those in the library. The derived classes inherit all of the behavior and functionality of their base classes, but you can extend them by adding new member variables and functions and modify the existing behavior by overriding inherited member functions.

Key Concepts

The following are central concepts in the application framework:

- At the heart of your application for Windows is an "application object."

  The application object manages a list of documents and dispatches commands to other objects in the program.
- The unit of data that the user works with is a document.

  The document maintains, loads, and stores its data.
- The user interacts with a document through a "view" on the document.

  A view is a window embedded in the client area of a frame window. It displays its document's data and takes mouse and keyboard input, which it translates into selection and editing actions.
- Objects in the user interface, such as menus and buttons, send commands to the documents, views, and other objects in the application. Those objects carry out the commands.

Figure 4.1 shows the relationship between a document and its view.

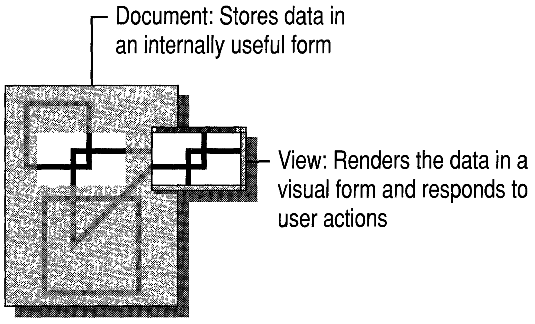

Document: Stores data in an internally useful form
View: Renders the data in a visual form and responds to user actions

Figure 4.1 Document and View

Working with the Framework

Your main tasks in using the framework are:

- Defining your application's data in its document class(es).
- Defining how the user views and interacts with the data inside a window.
- Connecting menus, buttons, and other user-interface objects to commands, then defining handler functions to carry out the commands.

The general process is described later in this chapter.

The Partnership

Your work with the Microsoft Foundation classes is a partnership, and your part is the source code that you add. This includes code to:

- Declare and implement the data structure of a document.
- Serialize the document's data so it persists from one work session to the next, typically by writing it to and reading it from a file.
- Display the data in a view.
- Process keyboard and mouse-related messages from Windows.
- Handle commands from menus and toolbar buttons.
- Enhance the printing, scrolling, and window-splitting capabilities you get from the framework.

In addition to the source code that implements your application's functionality, you're responsible for:

- Using Visual C++ to create and edit resources that define the user-interface elements of your program.
- Optionally, creating the rich-text format (RTF) files containing help topics for context-sensitive help.

The role of the framework in this partnership is to provide all of the many features detailed earlier in this chapter.

Benefits

The Microsoft Foundation Class Library (MFC) provides a thorough foundation that allows you to spend most of your programming effort writing the code that handles your data rather than reinventing the graphical user interface. For more information about the application framework, see Chapter 1 in *Programming with the Microsoft Foundation Class Library.*

Figure 4.2 shows schematically how your code fits into the framework.

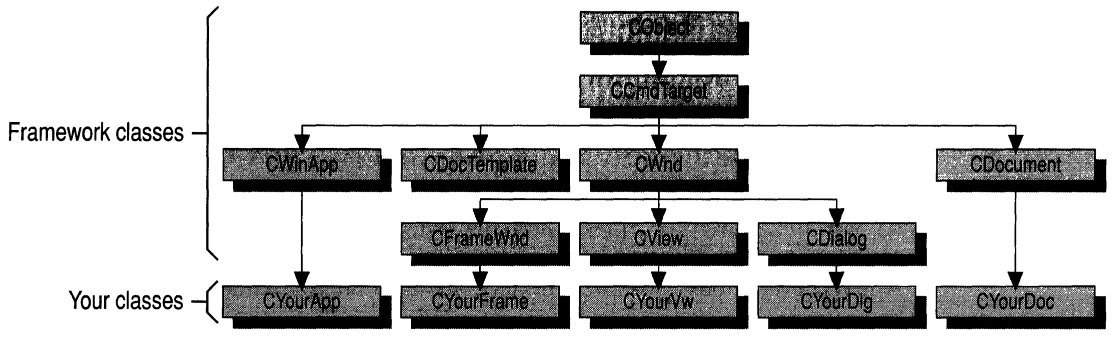


Figure 4.2 Your Code in the Application Framework

Because the framework provides so much standard functionality, it's easy to write applications that follow the recommendations of *The Windows Interface: An Application Design Guide.* At the same time, the framework's flexibility and extensibility don't lock you into *Design Guide* conformance, although deviating from the standard may take a little more work.

The Development Process

Developing a Visual C++ application can be broken into two stages: creating the application and developing the application. In both stages you will use Visual C++, which incorporates a text editor, resource editors, project window, browse window, and debugger in a single integrated development environment. Although you can use Visual C++ as a stand-alone resource editor to read and generate resource files for standard Windows SDK program development, you'll want to use it most often with ClassWizard and AppWizard to take advantage of MFC.

Creating the Application

The first step in creating a Visual C++ application is to name a project. Projects are the cornerstone of Visual C++. A project refers to the source files and libraries that make up a program, as well as the compiler and linker commands that build the program. It is composed of a makefile (.MAK), which is compatible with the Microsoft Program Maintenance Utility (NMAKE), and a project configuration file (.VCP). A project is identified by its makefile; the makefile has the same base name as the project, with an .MAK extension. All sample programs have project files associated with them.

You can use projects from existing 32-bit projects in Visual C++, which converts them to Visual C++ projects automatically when they load. This does not give you access to compiler and linker options (other than Release versus Debug mode), but it does help you bring applications into Visual C++ quickly and build, run, and debug your existing applications.

Creating the Basic Application

After you name your project, use AppWizard to generate a set of application starter files. It is important to use AppWizard first during the development of a Visual C++ application because it creates source code that is compatible with ClassWizard.

By selecting options in AppWizard, you can create C++ source files for skeleton applications with differing levels of functionality. If you select all of the AppWizard options, generate a project, and then build it without adding a single line of code, you'll get an application for Windows with the following features:

- A multiple document interface (MDI).
- Menus and dialog boxes for opening and saving files, previewing print jobs, and printing.
- Support for object linking and embedding (OLE).
- Support for Help.
- A functional toolbar and status bar.
- ODBC support

When first exploring AppWizard, you might want to start with the default options.

AppWizard creates all the files required for a standard Visual C++ application, including source files, resource files, and a project file. Visual C++ then loads the project. You can immediately compile and link the files by choosing Build from the Project menu.

Developing the Application

The development stage of an application for Windows involves editing source and resource files, compiling and linking, testing, and debugging. These activities are repeated throughout a normal development cycle. There is an order involved, however, because you always create user-interface objects (resources) first, then use ClassWizard to create the code shell and the text editor to write the functional code.

Creating and Editing User-Interface Objects

Although AppWizard creates some basic user-interface objects (such as menus, a toolbar, and so on) when it generates the resource files, you will probably want to add user-interface objects of your own. To learn how to create and edit user-interface objects, see Chapter 4, "Working with Resources," in the *Visual C++ User's Guide*.

Connecting User-Interface Objects to Code

After creating the user-interface objects, the next step is to create the MFC code that supports them. You use ClassWizard to generate the message-handler functions and message maps for each user-interface object you create. ClassWizard lets you manage your source code easily by identifying the proper location to add code.

Visual C++ coordinates all the other tools in the development process by maintaining the project information. You not only edit files and create resources; you also manage the source code and build and debug the application. The next two sections describe the tools you use during application development.

Managing Your Source Code

ClassWizard and the browse window are source-code management tools that allow you to access your source code from a structured viewpoint.

As discussed earlier, ClassWizard keeps track of all resource objects and member functions. It lets you immediately jump to the message-handler source code from ClassWizard so that you can edit it.

The browse window is another Visual C++ tool for managing source code. You can use the browse window to:

- Graphically display hierarchical class trees of derived or base classes.
- Graphically display all the functions that call, or are called by, a particular function.
- Display a list of source-code locations where references to a symbol are made and where a symbol is defined.
- Jump directly to definitions and references from list entries in the browse window or from a selected symbol in a source file.

You can use the browse window to show relationships between base and derived classes and between calling and called functions. You can also jump directly to source code simply by double-clicking a reference or definition in the browse window. Or, without using the browse window, you can select a symbol in a source file, jump to its definition or first reference, view all references to the symbol, and return to the original location, all using Search menu commands or shortcut keys.

Building, Running, and Debugging Your Application

Visual C++ helps you build, run, and debug your application with as little interruption as possible. You can build a project by:

- Choosing the Build or Rebuild All toolbar button:
  ,
- Choosing the Build or Rebuild All command from the Project menu.

When the build is complete, you can run the program in the integrated Windows-hosted debugger (assuming it includes debug information) by:

- Choosing the Run toolbar button:
- Choosing the Go command from the Debug menu.

You can run the program outside the debugger by choosing Execute from the Project menu.

The debugger has many powerful features, including:

- "Just-In-Time" debugging, which enables you to open a debugging session when an error occurs instead of closing your application.
- Breakpoints for breaking a program at a location, on an expression evaluation, or on a Windows message or class of messages.
- A QuickWatch dialog box for examining and changing variable values.
- A Watch window for examining specific variables and expressions.
- A Locals window for examining local variables.
- A Memory window for examining memory contents.
- A Registers window for examining and changing hardware register values.
- Tracing commands to step over, step into, or step out of functions.
- Multithreaded debugging capabilities.
- Structured exception handling.
- Mixed source and assembly listings and assembly-line tracing.

To use the debugger, you set breakpoints and run the application in a debug session. When the debugger reaches a breakpoint, you have several options. You can examine variables or expressions using the Watch window, the Locals window, or the QuickWatch dialog box. Or you can single-step through the code, choosing to step over or trace into functions that are encountered.

For More Information

The topics introduced in this chapter are covered in detail in a number of places in the Visual C++ documentation. Please refer to the following documentation for more information:

- Chapter 5, "A Test Drive of Visual C++," contains a quick tutorial that walks you through development of a sample Visual C++ program.
- Chapters 6 through 27 of this guide contain comprehensive tutorials on developing a Visual C++ application.

- Part 2 of *Programming with the Microsoft Foundation Class Library* covers conceptual information on using the Microsoft Foundation classes.
- Chapters 1 through 16 of the *Visual C++ User's Guide* describe how to use general Visual C++ features.
- Chapters 1 and 12 of the *Visual C++ User's Guide* describe AppWizard and ClassWizard, respectively, in detail.
- Chapter 17 of the *Visual C++ User's Guide* describes how to customize your working environment.
- Chapters 18–21 of the *Visual C++ User's Guide* provide reference information about toolbars, keyboard shortcuts, and compiler and linker options.

PART 2

Tutorials

CHAPTER 5

A Test Drive of Visual C++

This chapter takes you through a development session using Visual C++ tools to create, build, and debug a Microsoft Foundation Class Library (MFC) C++ application. This tutorial demonstrates the core Visual C++ development tools and the general development process. You can probably finish the entire tutorial in about an hour.

This tutorial doesn't teach you about MFC itself, but it does give you a tour of the product and shows you how easy it is to get started with Visual C++ programming. For a more detailed tutorial, see the Scribble tutorial beginning in Chapter 6. If you are unfamiliar with the tools and overall development process, it may be helpful to read Chapter 4, "Developing a Microsoft Visual C++ Application," in this guide before starting this tutorial.

The tutorial includes three sections: "Developing an Application," "Browsing the Application," and "Debugging the Application." The first section develops the application that is used by the other two sections.

Developing an Application This section demonstrates how to use AppWizard to generate a set of skeleton files for an application. It then describes how to create Windows-based resources and use ClassWizard to create and manage the code that supports these resources. When you complete this section, you will have a completed Visual C++ application.

Browsing the Application This section shows you how to use the source browser to examine class information in the Browse window or to jump to definitions and references from a symbol in a source file.

Debugging the Application In this section, you use the integrated debugger to set a breakpoint, examine variables, and trace through code.

Developing an Application

The application you'll create (called Myapp) is a subset of the VIEWEX sample application provided with the MFC samples. Myapp lets you open new child windows, each displaying an initial message, in a Multiple Document Interface (MDI) application. When you choose Change Text from the Edit menu in Myapp, a dialog box opens and allows you to change the text of the message in the currently active child window.

Myapp starts as a default application created by AppWizard. You'll add eight lines of code to this starter application to print a message when each new document window (MDI child window) is opened. Then, you'll create a dialog box for editing the message and add a menu command for opening the dialog box.

Running AppWizard

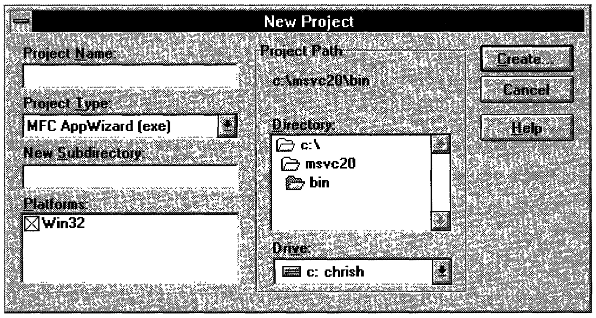
The first step in creating an MFC Visual C++ application is to use AppWizard to create the starter application.

▶ **To generate a set of application starter files using AppWizard**

1. Open Visual C++.
2. From the File menu, choose New.

   The New dialog box appears.
3. In the New list, select Project, then choose OK.

   The New Project dialog box appears (see Figure 5.1). The dialog box requires that you enter a project name and select a project location.



Project Name:

Project Type: MFC AppWizard (exe)

New Subdirectory:

Platforms: ☒ Win32

Project Path: c:\msvc20\bin

Directory: c:\ msvc20 bin

Drive: c: chrish

Figure 5.1 The New Project Dialog Box

4. In the Project Name box, type **myapp**.

   By default, Visual C++ creates the MYAPP directory as a subdirectory of the current directory (this tutorial uses C:\MSVC20\BIN as the current directory). To change the directory under which Visual C++ creates the MYAPP directory, double-click the directory you want to select.

Although Myapp will create the default MFC AppWizard (exe) project type, you could create any of the additional following types by selecting one from the Project Type list:

- MFC AppWizard (dll)
- Application
- Dynamic-Link Library
- Console Application
- Static Library

5. Choose Create.

   AppWizard displays the Step 1 dialog box, which lists types of applications you can create.

   This tutorial requires the Multiple Documents application type, the default option. AppWizard also allows you to create Single Document and Dialog-based applications.

6. Choose Next.

   AppWizard displays the Step 2 Of 6 dialog box, which lists database support options.

   Use the default option, None, to specify that AppWizard create an application with no database support. AppWizard offers the following alternatives:

   - Only include header files
   - A database view, without file support
   - Both a database view and file support

7. Choose Next.

   AppWizard displays the Step 3 Of 6 dialog box, which lists OLE options.

   Use the default option, None, to specify that AppWizard should create an application with no support for compound documents. AppWizard offers these additional alternatives:

   - Container
   - Mini-Server
   - Full-Server
   - Both container and server

   You can also choose whether to include automation support in your application. Use the default option, No Automation, to specify that no support be included.

8. Choose Next.

   AppWizard displays the Step 4 Of 6 dialog box, which lists project options.

   Use the following default project options:

   - Dockable Toolbar
   - Initial Status Bar
   - Printing and Print Preview
   - Use 3D Controls

   AppWizard also offers you the choice of creating context sensitive help. You will not create context sensitive help during this tutorial. You can also specify the number of files your application keeps on its "most recently used" (MRU) list. For this tutorial, use the default number, 4.

   From this dialog box, you can open the Advanced Options dialog box to change document template strings and specify the characteristics of window frames for your application. You will not specify advanced options in this tutorial.

9. Choose Next.

   AppWizard displays the Step 5 Of 6 dialog box, which lists three more project options.

   Use the default options, Yes, Please for source comments; Visual C++ Makefile; and Use MFC In A Shared DLL.

10. Choose Next.

    In the Step 6 Of 6 dialog box, AppWizard displays the names of classes and files in the starter application. (AppWizard derives a set of names based on the project name you type.) For this tutorial, you should not change the classes or files.

11. Choose Finish.

    AppWizard displays the New Project Information dialog box to confirm the type of application, the classes, and the features of the application it is about to create.

12. Choose OK to confirm this selection.

    AppWizard generates the starter files, creates a project, and loads it.

    A project consists of the names and locations of source files used to build your application, the settings of tools used to build the application, and the look and organization of the Visual C++ workspace you use to build your application. The project window displays a graph of logical relations among the files used to build your project (see Figure 5.2).

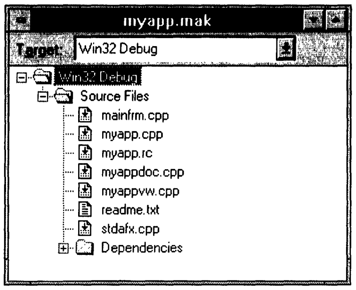


Figure 5.2 The Project Window

Because this is a new project, Visual C++ scans through all the project files to create an internal list of dependencies. This list determines which files must be recompiled when you build the project. For example, if a source file includes a header file and you edit the header file, the source file must be recompiled.

Building and Running Myapp

You may want to see just how much functionality you get with a basic AppWizard starter project created with default options. In this optional step, you'll build and run Myapp.

▶ **To build and run the application**

1. Click the Build toolbar button (or choose Build MYAPP.EXE from the Project menu).

   Toolbar:

   As Myapp is being built, the Output window appears and you'll see output from each of the tools—the compiler and linker, for instance—as the build progresses.

   By default, Visual C++ builds a version of Myapp which includes symbolic debugging information.

2. From the Project menu, choose Execute MYAPP.EXE.

   Myapp (see Figure 5.3) is an MDI application that contains a toolbar and a status bar that displays prompt messages. Among other features are a File menu with Open and Save commands (automatically hooked up to appropriate dialog boxes), a Window menu to arrange child windows opened from the New command on the File menu, and an About box on the Help menu.

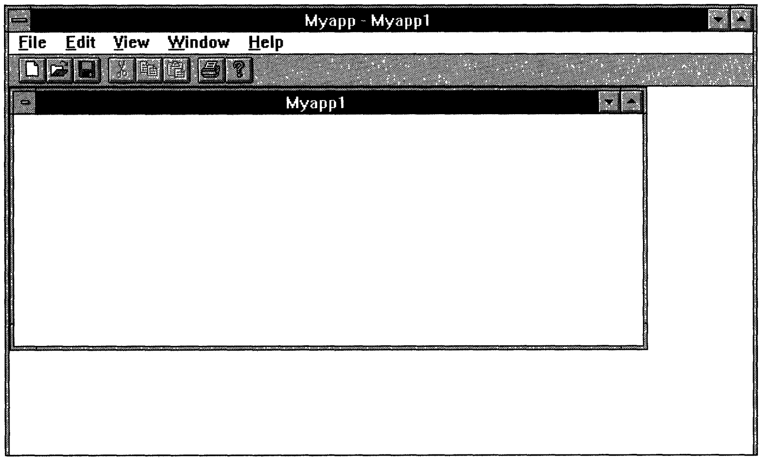


Figure 5.3 The Default AppWizard Application

3. From the File menu, choose Exit to close Myapp.

Modifying the Application

In this step, you'll add a few lines of code to the application to initialize and display a message string. In MFC version 3.0, the data is stored in document classes and displayed using view classes.

Note Every line of code to be added in this tutorial is indicated by an arrow in the left margin.

First, you'll use the following procedure to open MYAPPDOC.H and add a member variable to the class `CMyappDoc`.

▶ **To edit MYAPPDOC.H**

1. In the Project window, titled MYAPP.MAK, double-click the Dependencies icon (see Figure 5.4).

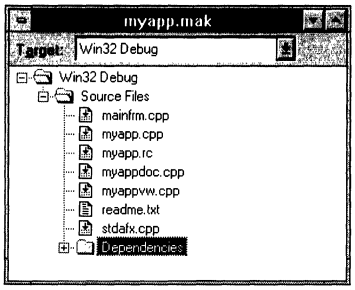


Figure 5.4 The Dependencies Icon in the Project Window

A list of files appears.

2. In the files list, double-click MYAPPDOC.H.

   The file appears in the text editor. AppWizard constructed the files MYAPPDOC.CPP and MYAPPDOC.H, derived from the project name you entered, for the document class `CMyappDoc`.

3. Find the `CMyappDoc` class declaration, and add the `m_strData` declaration in the public section:

```
// Attributes
public:
▶    CString m_strData;  // add for tutorial
```

4. Choose the Save toolbar button to save the file.

   Toolbar: 💾

5. From the File menu, choose the Close command to close the MYAPPDOC.H file.

Next, you'll edit MYAPPDOC.CPP to initialize the member variable `m_strData` for each new document window:

▶ To edit MYAPPDOC.CPP

1. In the Project window, double-click MYAPPDOC.CPP to open the file.
2. Find the `CMyappDoc::OnNewDocument` function, and add the marked line following the comments:

```
BOOL CMyappDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
▶   m_strData = "Sample Data String";
    return TRUE;
}
```

3. Choose the Save toolbar button to save the file.

   Toolbar:

4. From the File menu, choose Close to close the MYAPPDOC.CPP file.

Finally, you need to edit MYAPPVW.CPP to display the text.

▶ **To edit MYAPPVW.CPP**

1. In the Project window, double-click MYAPPVW.CPP to open the file.
2. Find the `CMyappView::OnDraw` function and add the specified lines following the TODO comment:

```
  void CMyappView::OnDraw(CDC* pDC)
  {
          CMyappDoc* pDoc = GetDocument();
          ASSERT_VAILD(pDoc);
        // TODO: add draw code for native data here
▶     CRect rect;
▶     GetClientRect(rect);
▶     pDC->SetTextAlign(TA_BASELINE | TA_CENTER);
▶     pDC->SetBkMode(TRANSPARENT);
▶     pDC->TextOut(rect.Width() / 2, rect.Height() / 2,

▶             pDoc->m_strData, pDoc->m_strData.GetLength());
      }
```

3. Choose the Save toolbar button to save the file.

   Toolbar:

At this point, if you build and run the application, you'll see that each new MDI window you create displays the string "Sample Data String."

▶ **To build and run the application**

1. Choose the Build toolbar button (or choose Build MYAPP.EXE from the Project menu).

   Toolbar:

   If necessary, correct any typing errors you might have made that cause syntax errors and repeat step 1. When there are no errors reported, continue.

2. To run Myapp, choose Execute MYAPP.EXE from the Project menu.

Adding User-Interface Objects to Myapp

In this step, you design a new dialog box and use ClassWizard to generate the code that supports it.

Creating a Dialog Box

In this step you create a new dialog box and add an edit control and a static text control to it. The dialog box will let the user change the message that is displayed in the active child window.

▶ **To create Myapp's dialog box**

1. If the Myapp project window is not open, choose Open from the File menu and double-click MYAPP.MAK in the File Name list.
2. From the Resource menu, choose New.

   The New Resource dialog box appears.
3. In the Resource Type list, select Dialog and then choose OK.

   Visual C++ opens the dialog editor. The editor displays a dialog box. The OK and Cancel buttons are already added for you.
4. From the Controls toolbar, drag the edit control to the middle of the dialog box (see Figure 5.5).

   If the Controls toolbar did not appear when you opened the dialog editor, choose the Toolbars command from the Tools menu, and then select the Controls box.

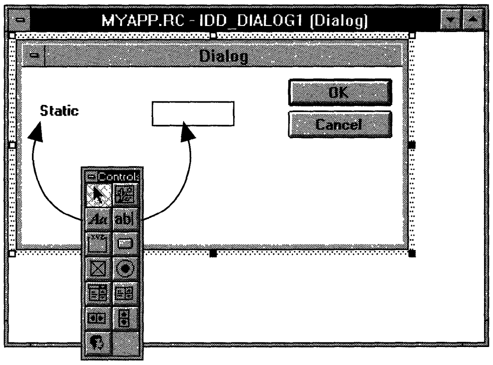


Figure 5.5 Dragging and Dropping Controls in the Dialog Editor

5. From the Controls toolbar, drag the static text control to the left of the edit control.

6. In the static text control, type **Edit**.

   Notice that as soon as you start to type, the Text Properties page appears. When you click the dialog box window again, the property page disappears (unless you choose to pin it open using the pushpin at the top left corner of the property page).

7. Now double-click the title (Dialog) and, in the Caption box of the Dialog Properties page, type **Change Text** in place of **Dialog**.

8. Choose the Save toolbar button to save your work.

   Toolbar:

Creating a Dialog Class and Adding a Member Variable

You can now use ClassWizard to connect this new dialog box to MFC framework code.

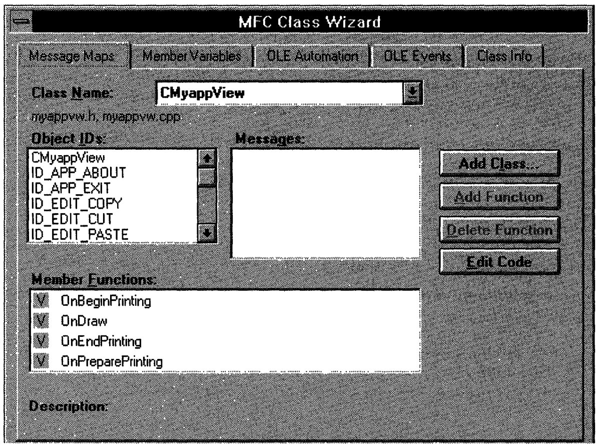
▶ **To create a dialog class**

1. Make sure the dialog box from the previous procedure is still open.

2. Click the ClassWizard toolbar button to open ClassWizard (or choose ClassWizard from the Project menu.)

   Toolbar:

   The MFC Class Wizard dialog box appears (see Figure 5.6).



Class Name: CMyappView

myappvw.h, myappvw.cpp

Object IDs:

CMyappView
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_COPY
ID_EDIT_CUT
ID_EDIT_PASTE

Member Functions:

V OnBeginPrinting
V OnDraw
V OnEndPrinting
V OnPreparePrinting

Figure 5.6 The MFC ClassWizard Dialog Box

3. Choose the Add Class button.

   The Add Class dialog box appears (see Figure 5.7).

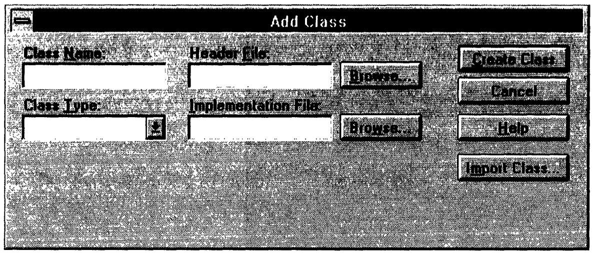


Figure 5.7 The Add Class Dialog Box

4. In the Class Name box, type **CEnterDlg**, the name of the new dialog class.

   Notice that the filenames "enterdlg.h" and "enterdlg.cpp" are automatically generated in the Header File and Implementation File boxes.

5. In the Class Type drop down list, select the **CDialog** class.
6. In the Dialog drop-down list, select the **IDD_Dialog1** identifier.
7. Choose the Create Class button.

   ClassWizard creates the header and implementation files named in step 4. The ENTERDLG.CPP file includes class declarations and skeleton implementation code. ClassWizard then returns you to the main ClassWizard dialog box.

   Don't close ClassWizard yet, because you'll be using it in the next procedure.

The dialog class now exists, but you still have to initialize the data for the edit control and retrieve data that has been entered into the edit control.

ClassWizard and the MFC framework make this easy with dialog data exchange (DDX). To implement DDX, you associate a member variable with a dialog box control. The framework transfers any initial values to the control when the dialog box is displayed. It also updates the member variable with any data the user has entered when the dialog box is dismissed.

You use ClassWizard to add a member variable to the `CEnterDlg` class for the dialog's edit control data.

▶ To add a member variable to CEnterDlg

1. With "CEnterDlg" showing in the Class Name drop-down list box, select the Member Variables tab of the MFC Class Wizard dialog box.

   The control ID for the edit control (IDC_EDIT1) should be selected.

2. Choose the Add Variable button.

   The Add Member Variable dialog box appears (see Figure 5.8).

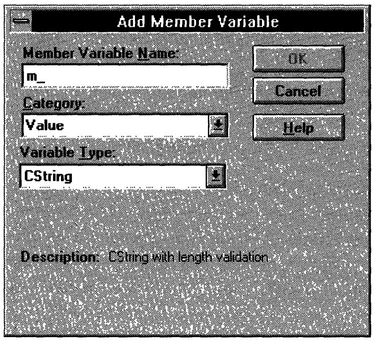


Figure 5.8 The Add Member Variable Dialog Box

3. In the Member Variable Name box, the prefix `m_` is displayed. Add **strInput** to this prefix to define the variable **m_strInput.**

   In the Property box, "Value" should be displayed. In the Variable Type drop-down list box, "CString" should be displayed.
4. Choose OK to exit the Add Member Variable dialog box.
5. Choose OK to exit ClassWizard.

   ClassWizard creates the required code in the `CEnterDlg` class for the `m_strInput` member variable in the files ENTERDLG.H and ENTERDLG.CPP.
6. Double-click the Control menu in the dialog editor window to close it.

Adding a Menu Item

In this step you add a new menu item and connect it to code by using ClassWizard. You then write the code that will display the new dialog box and handle its data when a user chooses the new menu item.

▶ **To add a menu item**

1. Click the resource browser window, titled MYAPP.RC (MFC Resource Script), to bring it to the front (see Figure 5.9).

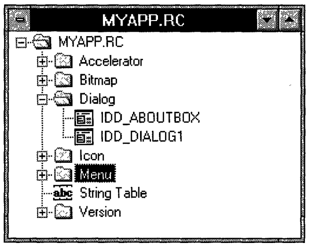


Figure 5.9 The Resource Browser Window

2. Double-click the Menu Folder icon.
3. Double-click IDR_MYAPPTYPE.

   The menu editor window opens.
4. Click the Edit menu in this window and the menu appears.
5. Select the new-item box at the bottom of the menu (below Paste), and type

   ```
   &Change Text...
   ```

   As soon as you begin to type, the property page for that menu item appears and the text you type appears in the Caption box.

   Notice that the ID box is empty. Leave it empty and Visual C++ will generate an ID for the menu item when you close the property page. If you want to enter it manually in the ID box, type:

   ```
   ID_EDIT_CHANGETEXT
   ```

   (It may be necessary to type it in if you display the ID drop-down list.)
6. In the property page's Prompt box, type:

   ```
   Opens a dialog box to change window text
   ```

   This text automatically appears in the prompt section of the status bar whenever the menu command is highlighted by the user.
7. Press ENTER to close the property page for your new menu command.
8. Remove all menu commands above Change Text by selecting each and pressing the DEL key.
9. Choose the Save toolbar button to save your work.

   Toolbar:

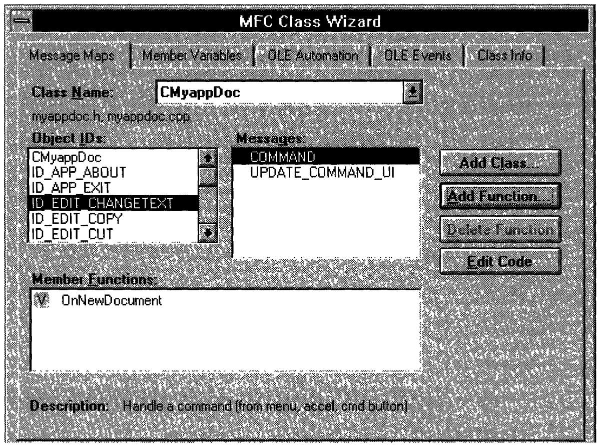
Creating the Menu's Message-Handler Code

Now you need to create the code that will be called when the user chooses the Change Text menu command. In the Microsoft Foundation Class Library, this means that you need both a message map entry and a member function associated with the menu command.

▶ **To add message-handler code for the menu command**

1. Choose the ClassWizard toolbar button to open ClassWizard.

   Toolbar:
2. If the Message Maps tab is not active, select it (see Figure 5.10).



Class Name: CMyappDoc
myappdoc.h, myappdoc.cpp

Object IDs:
CMyappDoc
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_CHANGETEXT
ID_EDIT_COPY
ID_EDIT_CUT

Messages:
COMMAND
UPDATE_COMMAND_UI

Member Functions:
V OnNewDocument

Description: Handle a command (from menu, accel, cmd button)

Figure 5.10 The Message Maps Tab

3. In the Class Name drop-down list, select `CMyappDoc`.

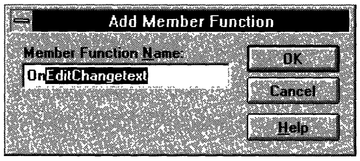
   You need to add the message handler code to this class.

4. In the Object IDs list, select the menu item's resource ID (**ID_EDIT_CHANGETEXT**).
5. In the Messages list, select **COMMAND**.

   (Do not select **UPDATE_COMMAND_UI**, which is used to dynamically update menu commands.)

6. Choose the Add Function button.

   The Add Member Function dialog box appears (see Figure 5.11). It allows you to change the name that ClassWizard has derived for the member function.



Member Function Name: OnEditChangetext

Figure 5.11 The Add Member Function Dialog Box

6. In the Add Member Function dialog box, accept the name `OnEditChangeText` by choosing the OK button.

Notice that the `OnEditChangeText` function is listed in the Member Functions box and that the **COMMAND** entry in the Messages box has a hand symbol next to it.

Don't close ClassWizard just yet, because you'll be using it in the next procedure.

Writing the Message-Handler Function

ClassWizard created a shell of the message-handler function. Now you need to write the body of the function. You can use ClassWizard to jump directly to the code in a source window for any member functions that have been added by ClassWizard. This procedure assumes that you have just finished the previous procedure and that the `OnEditChangetext` function is selected in ClassWizard.

▶ **To open a window at the message-handler function**

1. In ClassWizard, choose the Edit Code button.
2. Add the marked code after the TODO comments included in the MYAPPDOC.CPP file:

```
  void CMyappDoc::OnEditChangetext()
  {
      // TODO: Add your command handler code here
▶     CEnterDlg dlg;                  // create a CEnterDlg variable
▶     dlg.m_strInput = m_strData;      // initialize the edit string
▶     if (dlg.DoModal() != IDOK)       // open dialog box
▶         return;
▶     m_strData = dlg.m_strInput;      // retrieve edit string
▶     UpdateAllViews(NULL);           // general update
  }
```

3. Include the ENTERDLG.H header file at the top of MYAPPDOC.CPP:

```
  // myappdoc.cpp : implementation of the CMyappDoc class
  //

  #include "stdafx.h"
  #include "myapp.h"

  #include "myappdoc.h"
▶ #include "enterdlg.h"
```

4. Choose the Save toolbar button to save the file.

   Toolbar:

Building and Running the Application

The final step in this section of the tutorial is to build and run the Myapp application.

▶ **To build and run Myapp**

1. Click the Build toolbar button (or choose Build MYAPP.EXE from the Project menu).

   Toolbar:

   Assuming you've typed everything correctly and the build is successful, you can now run the program. If you have syntax errors, correct any typing mistakes and build the program again.

2. From the Project menu, choose Execute MYAPP.EXE.

   The application, shown in Figure 5.12, should have the following properties:

   - A default message "Sample Data String" appears for every new window opened
   - The Edit menu has one command, Change Text
   - When you select Change Text (or any command), a help message is displayed on the status bar.
   - When you open the Change Text dialog box, the default message is already in the text box. Any message you type in the text box appears in the active child window when you close the dialog box.

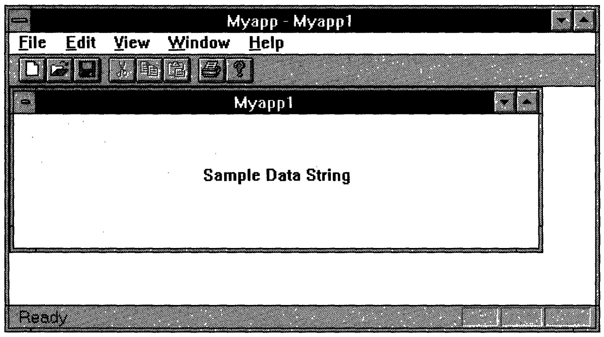


Figure 5.12 The Myapp Application

3. From the File menu, choose Exit to close the Myapp application.

Browsing the Application

This section of the tutorial uses the Myapp project you created in the preceding section. By default, generation of browser information is enabled and a browse information file is created when you build your project.

Examining Derived Classes and Members

In this step you'll use the Browse dialog box to display a Derived Class Graph. Then you'll pick a member variable of the class and jump to the first location in the source code where it is referenced.

▶ **To browse a Derived Class Graph**

1. From the Search menu, choose Browse.

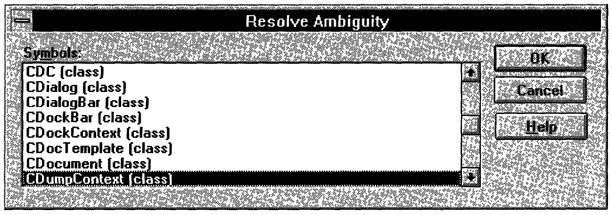
   The Browse dialog box appears.

2. In the Query On Name box, type **CD***.

   The asterisk (*) symbol is a wildcard that tells the browser to display a list of all classes that start with CD. Of course, you can also type the complete name of the class, but the wildcard is useful if you don't remember the exact name or don't want to type the full name.

3. From the Select Query list, double-click Derived Classes And Members.

   The Resolve Ambiguity dialog box appears (see Figure 5.13).



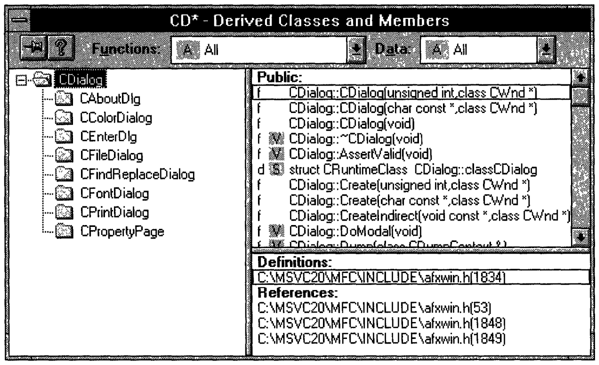
CDC [class]
CDialog [class]
CDialogBar [class]
CDockBar [class]
CDockContext [class]
CDocTemplate [class]
CDocument [class]
CDumpContext [class]

Figure 5.13 The Resolve Ambiguity Dialog Box

4. In the Symbols list, double-click **CDialog(class)**.

   Figure 5.14 shows the results of the query. Each icon in the left-hand pane represents a derived class. When you select a class, the member variables and member functions of the class are displayed in the upper-right pane, and a list of source locations where the class is defined and referenced appears in the lower-right pane.

   You can keep this browse window in view when it loses focus by clicking the pushpin at the top left of the window.

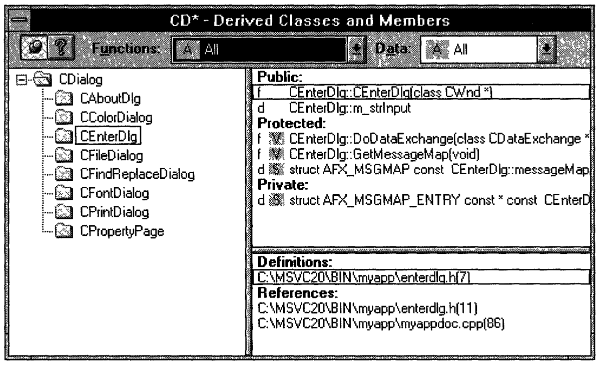


```
CDialog
  CAboutDlg
  CColorDialog
  CEnterDlg
  CFileDialog
  CFindReplaceDialog
  CFontDialog
  CPrintDialog
  CPropertyPage

Public:
f      CDialog::CDialog(unsigned int,class CWnd *)
f      CDialog::CDialog(char const *,class CWnd *)
f      CDialog::CDialog(void)
f  V   CDialog::~CDialog(void)
f  V   CDialog::AssertValid(void)
d  S   struct CRuntimeClass  CDialog::classCDialog
f      CDialog::Create(unsigned int,class CWnd *)
f      CDialog::Create(char const *,class CWnd *)
f      CDialog::CreateIndirect(void const *,class CWnd *)
f  V   CDialog::DoModal(void)

Definitions:
C:\MSVC20\MFC\INCLUDE\afxwin.h(1834)
References:
C:\MSVC20\MFC\INCLUDE\afxwin.h(53)
C:\MSVC20\MFC\INCLUDE\afxwin.h(1848)
C:\MSVC20\MFC\INCLUDE\afxwin.h(1849)
```

Figure 5.14 Derived Classes and Members of the CDialog Class

▶ To jump to a reference from the Browse window

1. Click the `CEnterDlg` icon in the left-hand pane. (See Figure 5.15.)



```
CDialog
  CAboutDlg
  CColorDialog
  CEnterDlg
  CFileDialog
  CFindReplaceDialog
  CFontDialog
  CPrintDialog
  CPropertyPage

Public:
f      CEnterDlg::CEnterDlg(class CWnd *)
d      CEnterDlg::m_strInput
Protected:
f  V   CEnterDlg::DoDataExchange(class CDataExchange *
f  V   CEnterDlg::GetMessageMap(void)
d  S   struct AFX_MSGMAP const  CEnterDlg::messageMap
Private:
d  S   struct AFX_MSGMAP_ENTRY const * const  CEnterD

Definitions:
C:\MSVC20\BIN\myapp\enterdlg.h(7)
References:
C:\MSVC20\BIN\myapp\enterdlg.h(11)
C:\MSVC20\BIN\myapp\myappdoc.cpp(86)
```

Figure 5.15 Jumping to a Reference from the Browse Window

`CEnterDlg` is the class you added, using ClassWizard, to handle the Change Text dialog box.

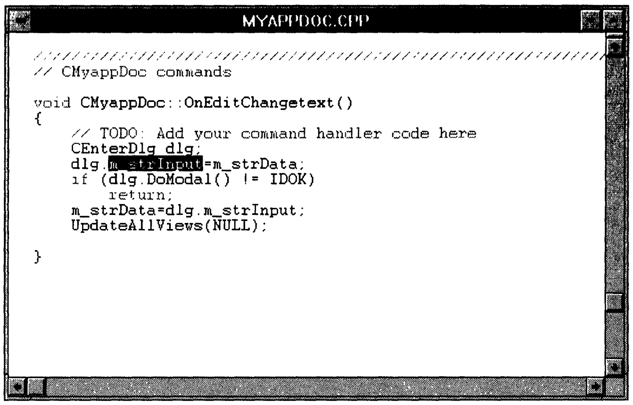
In the upper-right pane, you will see a list of functions and data symbols associated with the class, including the member variable you added to the class using ClassWizard (`CEnterDlg::m_strInput`).

2. Click `CEnterDlg::m_strInput` in the upper-right pane.

   A list of definitions and references for this member variable appears in the lower-right pane.

3. In the lower-right pane, double-click the first reference in the list of references: `c:\msvc20\bin\myapp\myappdoc.cpp`

   A source window, shown in Figure 5.16, opens on the corresponding file at the location of the first reference to `CEnterDlg::m_strInput`.



```
/////////////////////////////////////////////////////////////////////////////
// CMyappDoc commands

void CMyappDoc::OnEditChangetext()
{
    // TODO: Add your command handler code here
    CEnterDlg dlg;
    dlg.m_strInput=m_strData;
    if (dlg.DoModal() != IDOK)
        return;
    m_strData=dlg.m_strInput;
    UpdateAllViews(NULL);

}
```

Figure 5.16 The First Reference

If you want to continue browsing references or definitions, you can:

- Choose Next Reference from the Search menu to jump to the next reference.
- Choose Go To Definition from the Search menu to jump to the definition.
- Select another reference or definition from the lower right pane of the browse window.

Note that you can always use the definitions and references commands on the Search menu without opening the Browse dialog box. For example, put the insertion point on any symbol in a source file and press SHIFT+F11 (Go To Reference from the Search menu) and you jump to the first reference of that symbol, or press F11 and you jump to the symbol's definition. You don't need to open the Browse dialog box to use these commands since the browse information file (.BSC) loads automatically.

Examining a Base Class Graph

It is often helpful, especially when learning the architecture of MFC, to view the inheritance path of a class—that is, to view its base class, the base class of its base class, and so on. Base class graphs are usually composed of a single list of class nodes unless there is multiple inheritance somewhere in the hierarchy. Also, base class graphs are arranged so that each base class appears to the right of its derived class.

▶ To browse a Base Class Graph

1. From the Search menu, choose Browse.

   The Browse dialog box appears
2. In the Query On Name box, type **CEnterDlg**.

3. From the Select Query list, double-click Base Class Graph.

   The Base Class Graph window appears, showing the line of inheritance for the `CEnterDlg` class. The first node beneath `CEnterDlg` is **CDialog**, the framework class from which `CEnterDlg` is derived.
4. Click on the plus sign to the left of the **CDialog** icon.

   It expands one level to show its base class, **CWnd**.
5. Click on the plus sign to the left of the **CWnd** icon.

   It expands one level to show its base class, **CCmdTarget**.
6. Optionally, close the Base Class Graph window.

Debugging the Application

This section assumes you have loaded and built a debug version of the Myapp project as described in the first section of this tutorial.

Setting and Running to a Breakpoint

There are two ways to set a breakpoint in the debugger. You can use the Toggle Breakpoints toolbar button at a location in a source file, or you can use the Breakpoints dialog box, which allows more complex breakpoint expressions. The following procedure demonstrates how to use the toolbar to set a breakpoint.

▶ **To set a breakpoint and run Myapp to the breakpoint**

1. Open the MYAPPDOC.CPP file.
2. Move to the end of the file (CTRL+END).
3. Move the insertion point to the following line in the `CMyappDoc::OnEditChangetext` function:

```
m_strData = dlg.m_strInput;  // retrieve edit string
```

4. Click the Toggle Breakpoint button on the toolbar, or press F9, to set a breakpoint.

   Toolbar:

   Notice that a circle in the left margin indicates that the breakpoint is set. This breakpoint will be invoked whenever you close Myapp's Change Text dialog box during a debug session.
5. Click the Run toolbar button, or choose Go from the Debug menu.

   Toolbar:

   The Myapp application is loaded into the debugger and run.

6. From the Myapp Edit menu, choose Change Text.

   The Change Text dialog box appears.
7. In the Edit text box, type **New String** to replace **Sample Data String**.
8. Choose OK to close the Change Text dialog box.

   The execution line of the breakpoint you just set should appear.

Next, you will use the debugger to examine variables at the breakpoint. Don't close Myapp yet.

Examining Variables

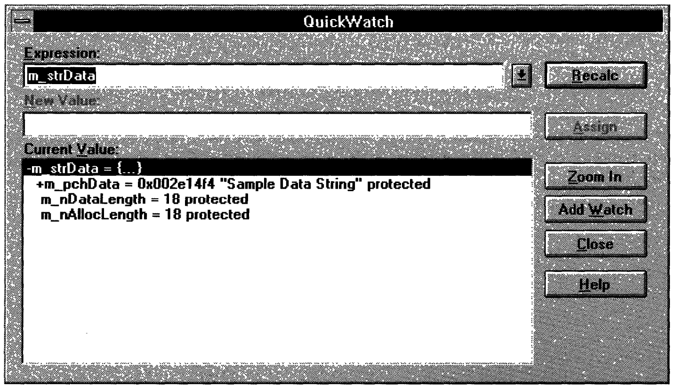
This step demonstrates how to use the QuickWatch dialog box to view the value of a variable and add the variable to the Watch window. It is assumed you have just finished the preceding procedure and execution is now paused at the breakpoint.

▶ **To open QuickWatch on a variable and add it to the Watch window**

1. On the breakpoint line, place the insertion point on the variable `m_strData`.
2. Press SHIFT+F9.

   The QuickWatch dialog box appears, displaying the contents of `m_strData` (see Figure 5.17). Since the line at the breakpoint has not yet executed, `m_strData` has not yet been assigned its new value.

   `m_strData` is a variable of type **CString** (a class). You can hide or display its member by double-clicking the variable name, which expands or contracts the structure. Two of the members are protected and therefore cannot be viewed.



Expression: m_strData

```
-m_strData = {...}
  +m_pchData = 0x002e14f4 "Sample Data String" protected
   m_nDataLength = 18 protected
   m_nAllocLength = 18 protected
```

Figure 5.17 The QuickWatch Dialog Box

3. Click the Add To Watch Window button to add this variable to the Watch window.

Whereas QuickWatch is handy for examining variables on the fly, you can use the Watch window to store variables and expressions so they are readily available at any time.

The next procedure opens the Watch window (if not already opened) and shows you another way to add variables to it.

▶ To open and use the Watch window

1. If you haven't already opened the Watch window, choose Watch from the Debug menu.

   The Watch window appears. You should see the `m_strData` variable that you just added from QuickWatch. You may want to arrange the Watch window and the source window containing the breakpoint so that you can see them both at the same time.
2. Switch to the MYAPPDOC.CPP source window and select `dlg.m_strInput` on the breakpoint line. From the Edit menu, choose Copy (CTRL+C).
3. Switch back to the Watch window.
4. Move the insertion point to the end of the prompt line in the Watch window. From the Edit menu, choose Paste (CTRL+V) to insert `dlg.m_strInput` into the Watch window.
5. Double-click `dlg.m_strInput` to expand the structure.

   The Watch window now contains both the **CString** variables that appear on both sides of the assignment operator in the line that contains the breakpoint:

   ```
   m_strData = dlg.m_strInput;
   ```
6. Choose the Step Over command from the Debug menu, or press F10.

   The execution point is now on the line following the breakpoint line. Notice that the value of `dlg.m_strInput` has been assigned to `m_strData`, since both variables now show the same values in the Watch window.
7. Choose the Run toolbar button to resume the Myapp program.

   Toolbar:

   The Myapp application appears again. Don't close Myapp if you want to continue the tutorial and learn about other trace commands.

Tracing Through Code

Although you used a couple of the trace commands in the last procedure, this procedure exercises them all. It assumes you have just finished the preceding procedure.

▶ **To trace through code**

1. In the Myapp application, open and close the ChangeText dialog box by choosing Change Text from the Edit menu.

   The execution should now be stopped at the breakpoint.

2. Open the file MYAPPVW.CPP and move the insertion point to the first line in the `CMyappView::OnDraw` function:

   ```
   CMyappDoc* pDoc = GetDocument();
   ```

3. From the Debug menu, choose Run To Cursor.

   The current insertion point location acts as a temporary breakpoint.

4. From the Debug menu, choose Step Into, or press F8.

   Notice that you traced into the opening brace of the `GetDocument` function in the same source file.

5. Now choose Step Into again.

   Execution is now at the first line of the `GetDocument` function:

   ```
   ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyappDoc)));
   ```

6. Choose Step Into again.

   This time you've traced into the MFC source code (the OBJCORE.CPP file). Although you may want to trace through this library code, you probably want to go back to the MYAPPVW.CPP file.

7. Choose the Step Out command from the Debug menu, or press SHIFT+F7.

   You have returned to the function call in MYAPPVW.CPP.

   In general, it's a good idea to step over most framework function calls. You can use the Step Out command to step out of a function that you don't want to step all the way through.

8. To conclude this tutorial, choose the Run button on the toolbar and exit Myapp.

   Toolbar:

   You should see a message box indicating that the process has terminated normally, meaning that the debugging session has finished and the program being debugged has exited.

For More Information

If you've completed all the procedures so far, you now have a good idea of how to use the tools provided with Visual C++ to create, browse, and debug MFC applications.

The tutorial has shown you how to use AppWizard to create the skeleton files, create resource objects, use ClassWizard to create code to support these objects and help navigate among them so you can add functional code. The tutorial has also shown you how to edit, browse, and debug code.

Use the information in the following books to help complete your understanding of MFC or any of the Visual C++ tools:

- *Programming with the Microsoft Foundation Class Library* for overview information.
- *Class Library Reference* for MFC classes and function reference.
- *Visual C++ User's Guide* for help on editing, building, browsing, and debugging, and using AppWizard, the resource editors, and ClassWizard.

CHAPTER 6

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. Something like Scribble 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) using the mouse, and save the image in a file.

By the end of the tutorial, Scribble has custom menus, a dialog box with automatic initialization and validation, printing and print preview, scrolling, splitter windows, context-sensitive Windows Help, and more. That's a fitting list of features for a GUI "Hello, World!" And, as you'll see, it's still quick to implement, considering the challenge.

Figure 6.1 shows what Scribble looks like on the screen.

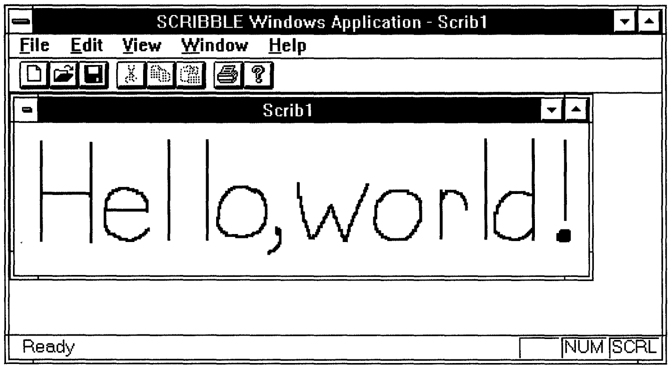

SCRIBBLE Windows Application - Scrib1
File Edit View Window Help
Scrib1
Hello, world!
Ready
NUM SCRL

Figure 6.1 Scribble in Action

Project Makefiles and STEP Directories for Scribble

Source code files, project files (also known as makefiles), and other necessary files for the tutorial are supplied in a group of subdirectories under the SAMPLES\MFC\SCRIBBLE directory. The tutorial develops the Scribble application in eight steps, with eight subdirectories (named STEP0 through STEP7) representing these steps. Chapters 6 to 15 describe how to develop the basic Scribble application (Steps 0 to 6). Chapter 16 describes how to convert Scribble to an OLE server (Step 7). Each subdirectory contains the files needed for one step. For convenience, Table 6.1 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 6.1 Tutorial Steps

Chapter	Step Completed	Content
7	0	Starter application (AppWizard)
8		Scribble's document
9	1	Scribble's view
10		Menus and toolbar (menu and graphics editors)
11	2	Handlers for commands (ClassWizard)
12	3	Dialog boxes (resource editors, ClassWizard)
13	4	Scrolling and splitting
14	5	Printing and print preview
15	6	Context-sensitive help
16	7	OLE server creation

For each version of Scribble, the project file, called SCRIBBLE.MAK, is stored in the appropriate subdirectory for the step. Use Table 6.1 to locate the right subdirectory for each chapter.

The Files You Work With

For both of the procedures that follow, you usually need to deal with only a few of the files:

- Document class files: SCRIBDOC.H and SCRIBDOC.CPP
- View class files: SCRIBVW.H and SCRIBVW.CPP

You may occasionally need to refer to (or edit) SCRIBBLE.H and SCRIBBLE.CPP, the application class files.

For chapters that use the resource editors (Chapters 9 to 11), you'll work with SCRIBBLE.RC, the application's resource file.

You may also occasionally want to examine the other files created by AppWizard and ClassWizard, but in most cases you won't need to alter them.

In Chapter 16, "Creating an OLE Server," you'll also work with the files for MFC classes that implement OLE server support.

Note For Chapter 7, simply follow the instructions for using AppWizard to create the skeleton 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 4, "Developing a Microsoft Visual C++ Application."

The Right Directory

If you're simply reading along with the tutorial without adding code, you can still compile Scribble at each step to see what it looks like and how it behaves. In this case, go to the STEP subdirectory for the step indicated at the beginning of a chapter. For example, in Chapter 8, go to the STEP1 subdirectory. Open the project by double-clicking the file SCRIBBLE.MAK from the File Open dialog box.

If you're working along, adding code as you read, compile the project in which you've been editing the files. You should already be in your working subdirectory and have the project open. In the case of Scribble, the working directory is MYSCRIB, and you create it in Chapter 7.

Setting Options

For Scribble, you'll normally want to use the default debug-mode setting. However, for your own project you'll want to be able to build a release version.

To select debug or release build options, from the Project window, choose the Debug or Release version in the Target box.

Once you've moved to the right directory and the correct options are set, you're ready to build Scribble. Choose one of the Build commands on the Project menu.

Chapter 7 begins the tutorial proper. You'll create a skeleton application with AppWizard. In later chapters, you'll build a more powerful Scribble application upon that skeleton.

CHAPTER 7

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 the Microsoft Foundation Class Library (MFC):

- Use AppWizard to create a skeleton application—a set of C++ starter files.
- Use Visual C++ resource editors to construct the user interface.
- Use ClassWizard and the text editor to add application-specific code to the starter files.
- Use Visual C++ to test and debug, then add more code.

To create a new Visual C++ project based on MFC, you'll use AppWizard, which creates all the code required to display the windows in which users will interact with your application. This code represents a starter application that you can compile and run immediately.

AppWizard speeds your work in beginning a new project. In seconds, it creates a set of Visual C++ files that declare skeletal versions of the classes that make up your application. Key parts of the code that implements these classes are supplied by AppWizard, based on the options you choose.

Once you've created the starter application with AppWizard, you'll complete the rest of the steps listed above. You'll use Visual C++ resource editors to construct the menus and other user-interface objects. Then you'll use ClassWizard to make connections between those objects and the code you write to respond to them. You'll also use Visual C++ to edit, compile, browse, and debug your source-code files.

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

This chapter shows you how to create a set of starter files for the Scribble application that is developed throughout the tutorial. These files contain skeletal code for several C++ classes—an "application class," a "document class," a "view class," and 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 the Microsoft Foundation Class Library*. You'll also learn more about them in Chapters 8 and 9 of this manual. Details about the created files are available in a text file (README.TXT) that is created along with the starter files. The contents of the starter files are discussed in later chapters as needed. For additional information about the starter files, see the article "AppWizard: Files Created" in Part 2 of *Programming with the Microsoft Foundation Class Library.*

Without adding a line of code, you can compile 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 the Windows operating system. The steps needed to compile and run the program are given in the sections "Compile the Starter Files" on page 94 and "Run the Starter Application" on page 95.

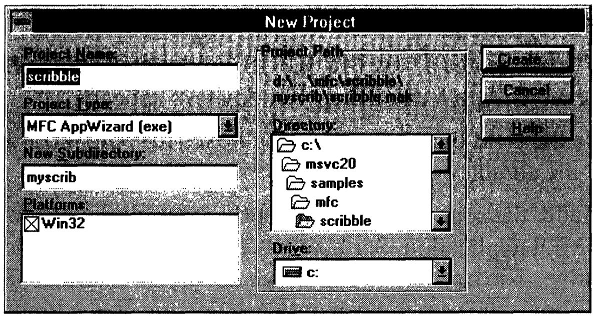
Chapters 8 and 9 show you how to add the application-specific code for Scribble, the sample application developed in the rest of the tutorial. Chapter 10 shows you how to construct Scribble's user interface with Visual C++. From Chapter 11 and the chapters that follow, you'll iteratively add more features to Scribble, then test, revise the interface, and so on.

This chapter covers step 0 of the tutorial. 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 subdirectory. 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. If you prefer, however, you can simply study the set of files in the SAMPLES\MFC\SCRIBBLE\STEP0 subdirectory, which were created by AppWizard.

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 lets you specify a number of options. Then it 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.

The following procedure describes how to enter the correct values for Scribble. This procedure applies equally to your own applications. Just change the names and other values as needed.



Project Name: scribble
Project Type: MFC AppWizard (exe)
New Subdirectory: myscrib
Platforms: ☒ Win32
Project Path: d:\...\mfc\scribble\myscrib\scribble.mak
Directory: c:\ msvc20 samples mfc scribble
Drive: c:

Figure 7.1 New Project Dialog Box

▶ **To create starter files for Scribble**

1. From the File menu, choose New.

   The New dialog box appears.

2. Select Project.

   The New Project dialog box (Figure 7.1) appears.

3. In the Project Name box, type **scribble**.

   The application's project file will be given this name: in this case, SCRIBBLE.MAK.

4. In the New Subdirectory box, delete "scribble" and type **myscrib**.

   This names the directory that will contain the project's files.

   AppWizard will create this directory if it doesn't exist. For Scribble, MYSCRIB is a new directory.

5. Specify the path to the project's subdirectory.

   Use the list box provided to navigate through the directories on the selected drive. As you navigate through the directory structure, the path listed in the dialog box changes to show where the named subdirectory (MYSCRIB) will be. When the path suits you, stop navigating.

   For Scribble, navigate to SAMPLES\MFC\SCRIBBLE (relative to your Visual C++ installation). Assuming your Visual C++ installation is in directory MSVC20 on drive C, the path should look like this in the dialog box:

   ```
   c:\msvc20\samples\mfc\scribble\myscrib\scribble.mak
   ```

   Note that in the Project box, you can choose several different kinds of projects to create, including MFC AppWizard (exe), the default and the option that Scribble uses; MFC AppWizard (dll); Application; and Dynamic Link Library. For more information on these choices, see Chapter 1, "Creating a New Application Using AppWizard," in the *Visual C++ User's Guide*.

6. Choose the Create button.

   AppWizard creates a new MYSCRIB subdirectory in the SCRIBBLE directory, and the MFC AppWizard-Step 1 dialog box appears.

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

   By default, the source code will support the multiple document interface (MDI). As an MDI application, Scribble lets the user open multiple documents at the same time. The alternative is a single document interface (SDI) application, which allows the user to open only one document at a time.

   For more information on the various options that appear in these dialog boxes, see Chapter 1, "Creating a New Application Using AppWizard," of the *Visual C++ User's Guide.*

8. In the AppWizard Step 4 dialog box, choose the Advanced button.

   The Advanced Options dialog box appears.

9. From the Document Template Strings tab, in the Document Type Name box, change "Scribb" to "Scrib."

   These are the characters (up to 6) used wherever Scribble's native document type is referred to. For example, the document type name is used to name the default file, SCRIB1.SCR, the Windows shell registration name ("Scrib"), and constants referred to in the code created by AppWizard, such as **IDR_SCRIBTYPE**.

   Type "scr" in the File Extension box. This extension is appended by default to the names of files that the user saves with Scribble. In the compiled Scribble application, "Scrib Files (*.scr)" appears in the List of Files of Type: box in the File Open and File Save As dialog boxes.

10. Choose Close.

11. Choose the Next button in the dialog boxes for AppWizard Steps 4 and 5 to accept the default options.

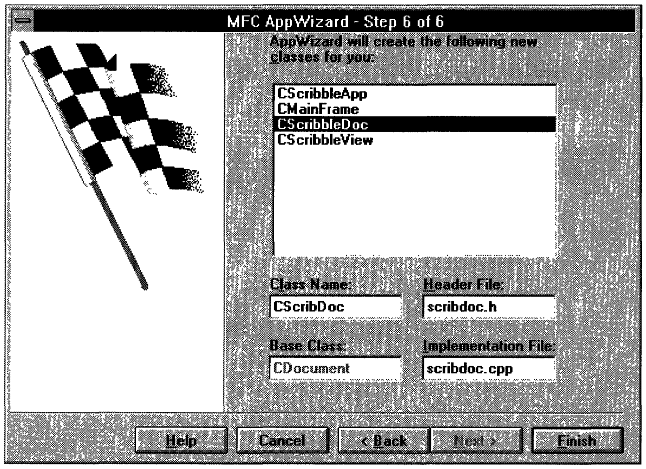
    These options on the Step 4 dialog box add code for printing and print preview and for supporting a toolbar. The options on the Step 5 dialog box supply comments throughout the files AppWizard creates to help you understand where you need to add your own code.

    For more information on the various options that appear in these dialog boxes, see Chapter 1, "Creating a New Application Using AppWizard," of the *Visual C++ User's Guide.*

12. In the AppWizard Step 6 dialog box, select class "CScribbleDoc" and change the class name from "CScribbleDoc" to "CScribDoc."

    In this and the next step, you check and modify class names and filenames. When the names are not dimmed, you can edit the names of your program's classes and the files. To edit the information for a class, select the class name in the box at the top of the AppWizard Step 6 dialog box, which is shown for the document class in Figure 7.2.

    For Scribble, some of the class names must be changed from the defaults that AppWizard suggests so that the names of your classes and files will be synchronized with those in the subdirectories provided for the Scribble steps. What you see on the screen as you work will also match the figures in the tutorial.


MFC AppWizard - Step 6 of 6

AppWizard will create the following new classes for you:

CScribbleApp
CMainFrame
CScribbleDoc
CScribbleView

Class Name: CScribDoc
Header File: scribdoc.h
Base Class: CDocument
Implementation File: scribdoc.cpp

Figure 7.2 The Document Class

13. Select class "CScribbleView" and edit its information as follows:

    Change the class name from "CScribbleView" to "CScribView."

    AppWizard provides satisfactory defaults for classes `CScribbleApp` and `CMainFrame`. You don't have to edit them for Scribble—although you might want to edit them for another application.

14. Choose the Finish button when you're done.

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

    AppWizard creates the specified subdirectory if it doesn't exist; then it creates all necessary files in the directory and opens the project.

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

Compile the Starter Files

The starter application you created provides the skeleton of a working application for the Windows operating system. When you compile the starter files—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. Of course, at this stage the windows have nothing in them. So far, Scribble doesn't scribble.

At this point, Scribble should be the currently open project.

▶ **To compile the starter application**

1. Make sure you've set up the environment as explained in Chapter 1, "Installing Microsoft Visual C++," in this manual.
2. From the Project menu, choose Build or Rebuild All as shown in Figure 7.3.
3. The starter application is built, producing the file SCRIBBLE.EXE in your new MYSCRIB subdirectory.

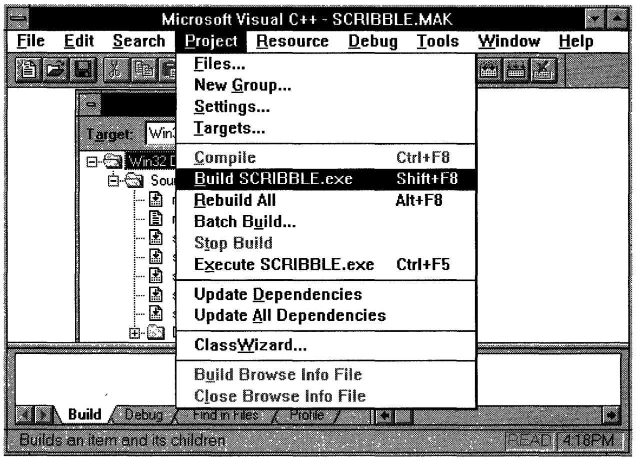


Figure 7.3 Compiling Scribble

Run the Starter Application

After you compile the starter application, you can run it from Visual C++, with or without breakpoints being set. You can also run the application from the Windows Program Manager.

▶ **To run Scribble for debugging (with breakpoints)**

1. From the Debug menu, choose Breakpoints and set any breakpoints you want in your code.
2. From the Debug menu, choose Go to run Scribble.

▶ **To run Scribble without debugging**

- From the Project menu, choose Execute.

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 7.4..

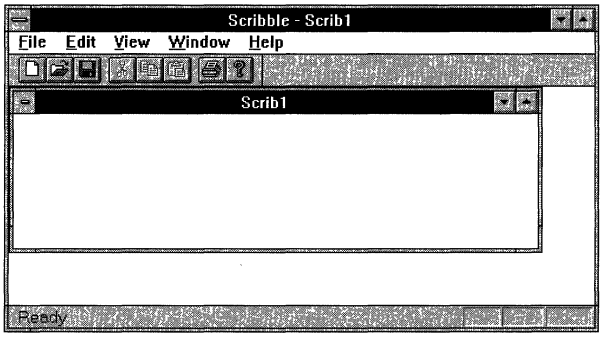


Figure 7.4 The Starter Application

The document window is empty, of course, because you've added no application-specific code yet. But you can move, resize, minimize, maximize, and close the document window and the application window. You can also use the New command on the File menu to open new windows. The Open, Save, and Save As commands are partially functional: at this point, they save empty files. You haven't added all of the code yet to support these commands. 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 Open and Save/Save As buttons do the same things as the corresponding menus. And the status bar at the bottom of the application window displays a description string when you move the mouse pointer over any menu command.

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

You'll undoubtedly want to examine the source code files you created. To orient you, AppWizard creates a text file, README.TXT, in your new application 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 the Microsoft Foundation Classes.*

For more information about using AppWizard, see Chapter 1, "Creating a New Application Using AppWizard," of the *Visual C++ User's Guide.*

CHAPTER 8

Creating the Document

In this chapter and the next, you'll add code to the starter application you created in Chapter 7 with AppWizard. By the end of Chapter 9, you can compile and run the Scribble program.

Figure 8.1 shows what the Scribble application developed in the tutorial will look like at the end of Chapter 9.

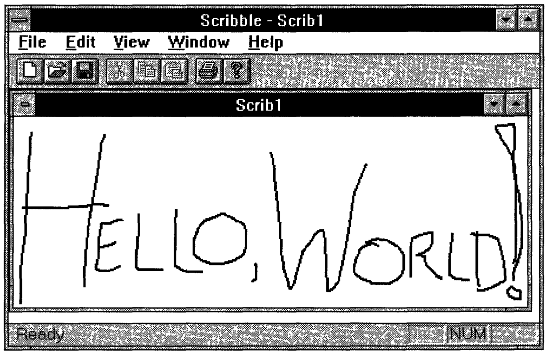


Figure 8.1 Scribble in Action

This chapter introduces documents and develops Scribble's document class, called `CScribDoc`, an application-specific class derived from class **CDocument**. Chapter 9 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. 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, context-sensitive help, and OLE server support.

Your tour of Scribble's code begins with the starter files created by AppWizard in the previous chapter. 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—`CScribDoc`, a class to contain and manage a list of strokes.
- Scribble's serialization code—code that implements writing and reading documents.

The code that you must add to fill out the framework in this chapter is in the following files: STDAFX.H, SCRIBBLE.H, SCRIBDOC.H, and SCRIBDOC.CPP.

This chapter and Chapter 9 cover step 1 of Scribble. If you want to work along, adding the code as you go, begin with the files you created with AppWizard in your MYSCRIB subdirectory in Chapter 7. At this point, your files should consist entirely of the set of generic starter files that AppWizard created. As you read this chapter, add or change all lines of code marked with the ▸ symbol in the left margin. At the end of Chapter 9,, your files should essentially resemble those in the SCRIBBLE\STEP1 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP1 subdirectory.

Note If you have trouble locating the correct place to add code, try looking at the corresponding source files in the subdirectory for the completed step. For this chapter and Chapter 9, use the SCRIBBLE\STEP1 subdirectory for this purpose.

If you want to preview Scribble, load the version in SCRIBBLE\STEP1 and choose Build on the Project menu. Run the program by choosing Execute from the Project menu.

Documents

At the heart of 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 the Microsoft Foundation Class Library (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 8.2 shows graphically the relationships between these key objects.

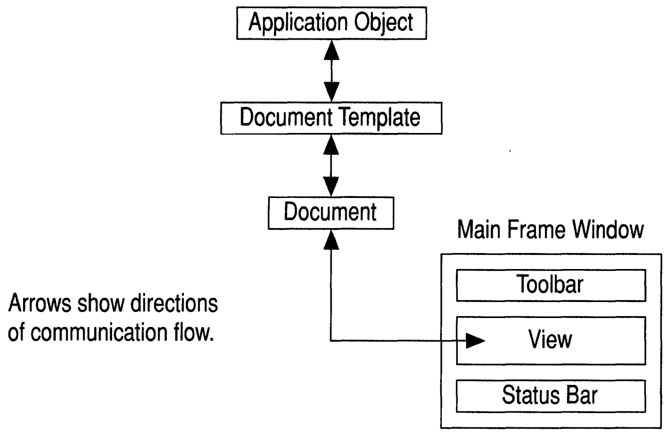


Figure 8.2 Objects in Scribble

Table 8.1 shows how the document and other objects are created and managed in a framework application.

Table 8.1 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 to the 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 8.3 shows the general relationship between a document and its view and frame window.

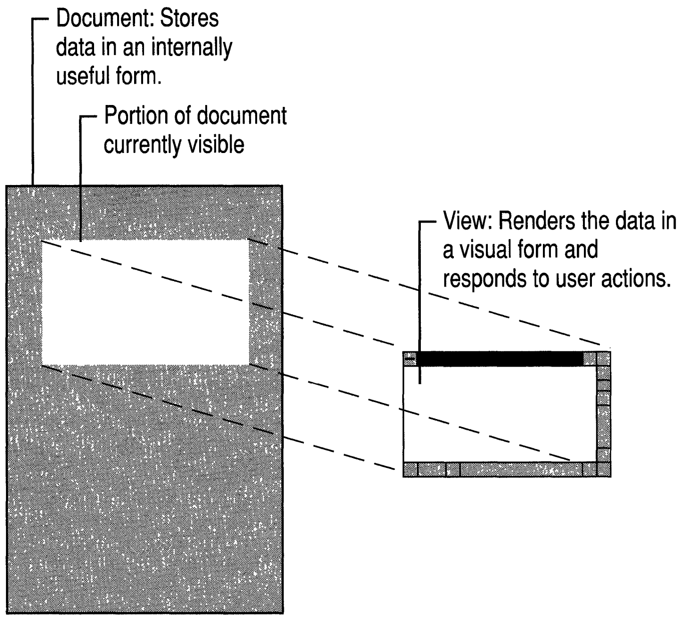

Document: Stores data in an internally useful form.
Portion of document currently visible
View: Renders the data in a visual form and responds to user actions.

Figure 8.3 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 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 8.3 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 and stores its data. The view obtains data from the document and displays it. Figure 8.4 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 the Microsoft Foundation Class Library*, as well as class **CDocument** in the *Class Library Reference*.

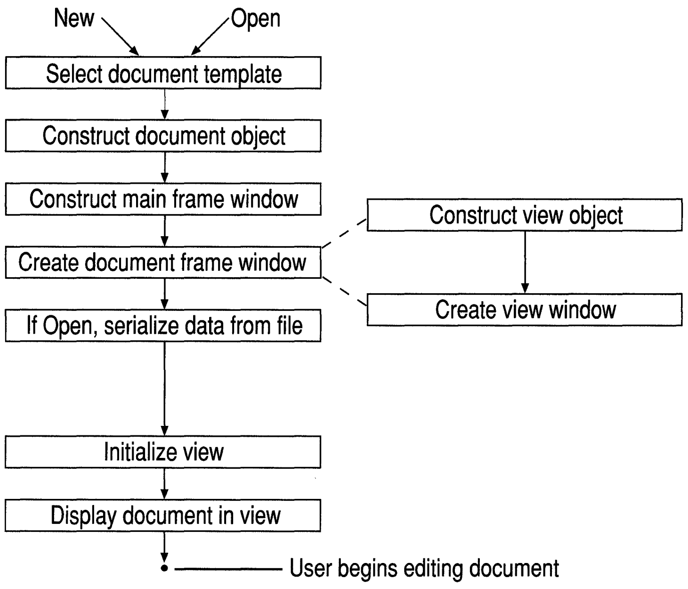

New
Open
Select document template
Construct document object
Construct main frame window
Create document frame window
If Open, serialize data from file
Construct view object
Create view window
Initialize view
Display document in view
User begins editing document

Figure 8.4 Creating a Document

Document/View Interaction

When the user modifies data through the view, the view notifies the document. In turn, the document tells all of its views (some documents 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 9.

You and the Document

Table 8.2 shows your responsibilities and those of the framework in implementing a document.

Table 8.2 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 CScribDoc

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 8.5 shows a single stroke drawn in a Scribble document's view.

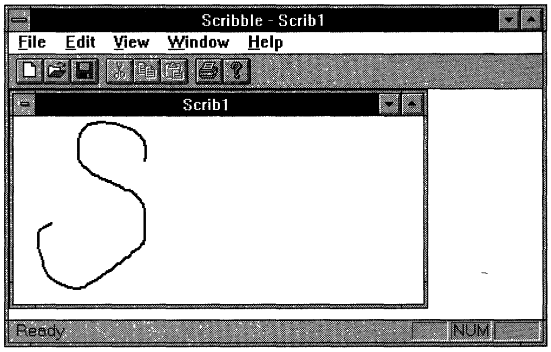


Figure 8.5 One Stroke in Scribble

Documents in Scribble are objects of class `CScribDoc`, which is derived from **CDocument**. `CScribDoc` has a member variable named `m_strokeList` for storing a list of strokes and member functions to manage the stroke list.

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

AppWizard writes a skeletal `CScribDoc` class for you, but you need to add a few things. In the following procedure, as in similar procedures throughout the tutorial, you'll add the indicated code to a named file, using the text editor. The files are those created by AppWizard. In the code listings below, the lines to add or change are always preceded by a few existing code lines or a comment to help you locate the right place in the files. To add code, start Visual C++, open the file named in the procedure, locate the place to add your lines, and type in the lines that are marked with the symbol ▶. For more information about editing source files, see the *Visual C++ User's Guide.*

To open a file in Visual C++, go the project window, which is labeled with the name of your make file. Implementation files, for example, .CPP and .RC files, can be found under the heading Source Files; other files are found under the heading Dependencies. Double-click the file you want to work in.

▶ **To add member declarations to Scribble's document class**

- Complete the declaration of class `CScribDoc` (in file SCRIBDOC.H). If you're working along, add the marked lines, which contain new Scribble-specific code, to the file.

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

  class CScribDoc : public CDocument
  {
  protected: // Create from serialization only
      CScribDoc( );
      DECLARE_DYNCREATE( CScribDoc )

  // Attributes
▶ protected:
▶     // The document keeps track of the current pen width on
▶     // behalf of all views. We'd like the user interface of
▶     // Scribble to be such that if the user chooses the Draw
▶     // Thick Line command, it will apply to all views, not just
▶     // the view that currently has the focus.

▶     UINT    m_nPenWidth;    // current user-selected width
▶     CPen    m_penCur;           // pen created according to
▶                          // user-selected pen style (width)
  public:
▶     CTypedPtrList<CObList, CStroke*> m_strokeList;
▶     CPen*   GetCurrentPen( ) { return &m_penCur; }
// Operations
public:
▶    CStroke* NewStroke( );

// Overrides
    // ClassWizard generate virtual function overrides
    //{{AFX_VIRTUAL(CScribDoc)
    public:
    virtual BOOL OnNewDocument( );
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CScribDoc( );
    virtual void Serialize( CArchive& ar ); // Overridden for
                                     // document i/o
#ifdef _DEBUG
    virtual   void AssertValid( ) const;
    virtual   void Dump( CDumpContext& dc ) const;
#endif

protected:
▶    void InitDocument( );

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

This code declares a C++ class that defines Scribble's documents. The member variables and functions provide the typical functionality of a document—they define and manipulate the document's data, serialize the data to and from files, and provide diagnostic assistance when you compile a debug version. (Notice the forward declaration of class `CStroke`, the class used to define the document's data structure. `CScribDoc` needs to know about `CStroke`. You'll learn about `CStroke` later in the chapter.)

Scribble uses two of several C++ collection template classes provided by MFC: **CTypedPtrList** and **CArray**. 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 put it into Scribble's precompiled header.

▶ **To add AFXTEMPL.H to the precompiled header**

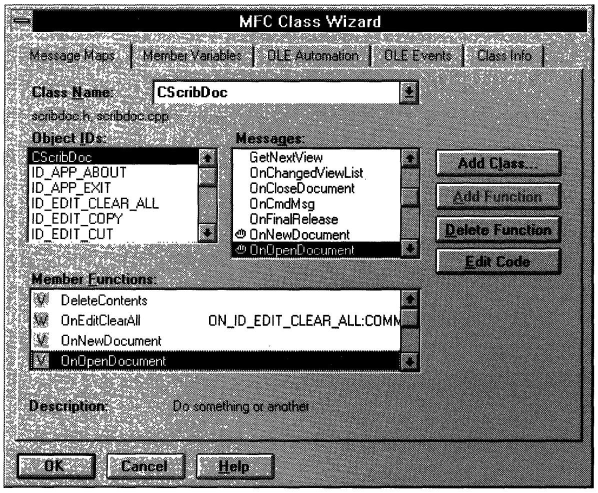
1. Open STDAFX.H, which was originally created by AppWizard to keep the list of header file to be precompiled.
2. Type **#include <afxtempl.h>**.

The final task in declaring the member functions for `CScribDoc` is to override **OnOpenDocument** and **DeleteContents**, which are needed for Scribble-specific initialization and clean-up. ClassWizard allows you to easily override virtual functions declared in a base class. ClassWizard lists and then lets you override the common virtual functions that appear above the `//Implementation` line in the header files. ClassWizard automatically adds the correct parameters and syntax for the definition of the member function that is overriden.

▶ **To override OnOpen Document and DeleteContents using ClassWizard**

1. From the Project menu, choose ClassWizard.
2. Choose the Message Maps tab.
3. In the Class Name box, select CScribDoc.
4. In the Object IDs box, select CScribDoc.
5. In the Messages box, select OnOpenDocument.
6. Choose the Add Function.

   Figure 8.6 shows how the Message Maps property page looks at this point. Note that the name of the overridden function appears in Member Functions box and that a gray glyph containing the letter "V" precedes the function name. Notice that AppWizard automatically provided an override for **OnNewDocument** when it created the skeleton for your application.



Class Name: CScribDoc
scribdoc.h, scribdoc.cpp

Object IDs:
CScribDoc
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_CLEAR_ALL
ID_EDIT_COPY
ID_EDIT_CUT

Messages:
GetNextView
OnChangedViewList
OnCloseDocument
OnCmdMsg
OnFinalRelease
OnNewDocument
OnOpenDocument

Member Functions:
V DeleteContents
W OnEditClearAll ON_ID_EDIT_CLEAR_ALL:COMM
V OnNewDocument
V OnOpenDocument

Description: Do something or another

Figure 8.6 Overriding OnNewDocument

7. Repeat steps 5 and 6 for **DeleteContents**.
8. Choose OK. Before choosing OK, you can choose the Edit Code button to examine the code or edit it. You will edit the code in "Initializing and Cleaning Up."

   ClassWizard writes the overridden functions into SCRIBDOC.H. The resulting code now looks like:

```
// Overrides
    // ClassWizard generate virtual function overrides
    //{{AFX_VIRTUAL(CScribDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
    virtual void DeleteContents();
    //}}AFX_VIRTUAL
```

   AppWizard added the implementation for `OnNewDocument`. ClassWizard has added the implementation for the `OnOpenDocument` and `DeleteContents`.

Table 8.3 describes the member variables of class `CScribDoc`.

Table 8.3 CScribDoc 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 classes, **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 8.4 describes the member functions.

Table 8.4 CScribDoc Member Functions

Member	Description
`CScribDoc, ~CScribDoc`	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.

Table 8.4 CScribDoc Member Functions *(continued)*

Member	Description
`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. Overriding 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" on page 113. For more information about **AssertValid** and **Dump**, see "Diagnostics" in *Programming with the Microsoft Foundation Class Library.* 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 8.7 shows Scribble's data structure schematically. Scribble uses an object of class **CPen** for drawing.

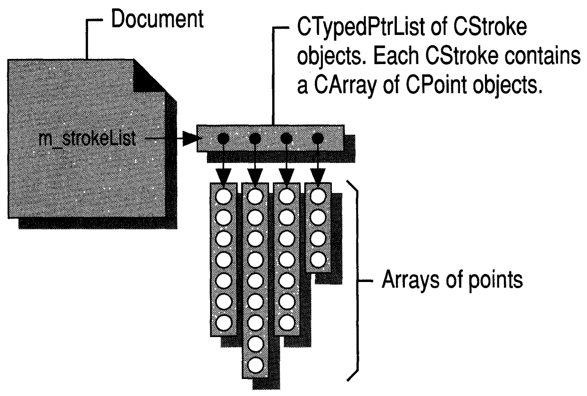

Document
CTypedPtrList of CStroke objects. Each CStroke contains a CArray of CPoint objects.
m_strokeList
Arrays of points

Figure 8.7 Scribble's m_strokeList Data Structure

Each stroke is stored in an object of class `CStroke`, Scribble's primary data structure. The whole drawing is a list of `CStroke` objects. `CStroke` is a new class, so you'll have to add its entire declaration to Scribble's source files.

▶ **To add the CStroke class**

- If you're working along, add the code marked below. The declaration for class `CStroke` follows that for class `CScribDoc` in file SCRIBDOC.H. Here's the declaration for class `CStroke`:

```
  // Declaration of class CScribDoc, then ...
▶ class CStroke : public CObject
▶ {
▶ public:
▶     CStroke( UINT nPenWidth );

▶ protected:
▶     CStroke( );
▶     DECLARE_SERIAL( CStroke )

▶ // Attributes
▶     UINT   m_nPenWidth;  // One width applies to entire stroke
▶     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. The member variables and functions 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.

Table 8.5 lists `CStroke`'s member variables.

Table 8.5 CStroke Data Members

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

Table 8.6 lists `CStroke`'s member functions.

Table 8.6 CStroke Member Functions

Member	Description
`CStroke`	The class defines two constructors, one protected and one public.
`DrawStroke`	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`	To assist the document in making its data persistent, typically on disk, `CStroke` also 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" on page 113.

Building and Storing Strokes

Your next step is to 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 the SCRIBDOC.CPP file:

```
  // Last line of CScribDoc code, then ...
▶ CStroke::CStroke()
▶ {
▶ // This empty constructor should be used by serialization only
▶ }
▶ CStroke::CStroke(UINT nPenWidth)
▶ {
▶     m_nPenWidth = nPenWidth;
▶ }
```

The first constructor, declared protected in SCRIBDOC.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. These member functions are used by the view object to draw the data.

Managing the Document

Typically, you must write code to (a) initialize a document's data members and (b) deallocate memory allocated for the data, release system resources, and perform other cleanup chores. When a new Scribble document is created, `CScribDoc` must create a pen for drawing new strokes. When a document is closed, the document must delete the stroke objects it has stored up.

Initializing and Cleaning Up

Because a document can be created with either the New command or the Open command on the File menu, `CScribDoc` 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`.

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`.

If you're working along, add the marked code to the indicated file. Because the constructor and destructor created by AppWizard are empty, the code isn't shown here.

▶ **To implement initialization for Scribble's documents**

1. Add a definition for the `InitDocument` member function to SCRIBDOC.CPP:

```
  // OnOpenDocument and DeleteContents, then ...
▶     void CScribDoc::InitDocument( )
▶     {
▶         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. Add the following to the override of **OnNewDocument** created by AppWizard in file SCRIBDOC.CPP:

```
  // Empty constructor and destructor, then ...
  BOOL CScribDoc::OnNewDocument( )
  {
      if( !CDocument::OnNewDocument( ) )
          return FALSE;
▶     InitDocument( );
      return TRUE;
  }
```

3. Finally, replace the ClassWizard skeleton version of **OnOpenDocument** in SCRIBDOC.CPP:

```
  // OnNewDocument, then ...
  BOOL CScribDoc::OnOpenDocument(LPCTSTR lpszPathName)
▶ {
▶     if( !CDocument::OnOpenDocument( lpszPathName ) )
▶         return FALSE;
▶     InitDocument( );
▶     return TRUE;
▶ }
```

   The two overrides call the base-class version of the function before performing application-specific initialization of the document.

   Because SCRIBDOC.CPP is a simple file, it is easy to find the implementations of `OnNewDocument` and `OnOpenDocument`. However, in more complicated files, you can use ClassWizard to locate the implementation of the overridden function. Open ClassWizard, select the member function you want in the Member Functions box, and choose Edit Code. The text editor will display the implementation for the function you chose.

▶ **To implement document cleanup**

- In file SCRIBDOC.CPP, replace the ClassWizard skeleton version of the **DeleteContents** member function of **CDocument**:

```
    // Empty constructor and destructors, then ...
    void CScribDoc::DeleteContents( )
    {
▶       while( !m_strokeList.IsEmpty( ) )
▶       {
▶           delete m_strokeList.RemoveHead( );
▶       }
        CDocument::DeleteContents( );
    }
```

`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.

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

Managing the Data

▶ **To implement document members for managing Scribble's data**

- Add the `NewStroke` member function to SCRIBDOC.CPP. `NewStroke` creates a new stroke object and adds it to the stroke list:

```
  // InitDocument function, then ...
▶ CStroke* CScribDoc::NewStroke( )
▶ {
▶     CStroke* pStrokeItem = new CStroke(m_nPenWidth);
▶     m_strokeList.AddTail( pStrokeItem );
▶     SetModifiedFlag( ); // Mark document as modified
▶                         // to confirm File Close.
▶     return pStrokeItem;
▶ }
```

`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 changes to the document, and it returns a pointer to the stroke.

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 the Microsoft Foundation Class Library.*

Serializing the Data

This section adds code to define file input/output for Scribble documents. The default I/O implementation in MFC is called "serialization" (see Figure 8.8). It provides a mechanism for making a document's data persistent between work sessions with the program. Given the code added here, serialization is automatic when the user chooses the Open, Save, or Save As commands from the File menu.

Note You don't have to write any code to process the Open, Save, and Save As commands on the File menu—such as putting up the dialog boxes. The framework supplies this code.

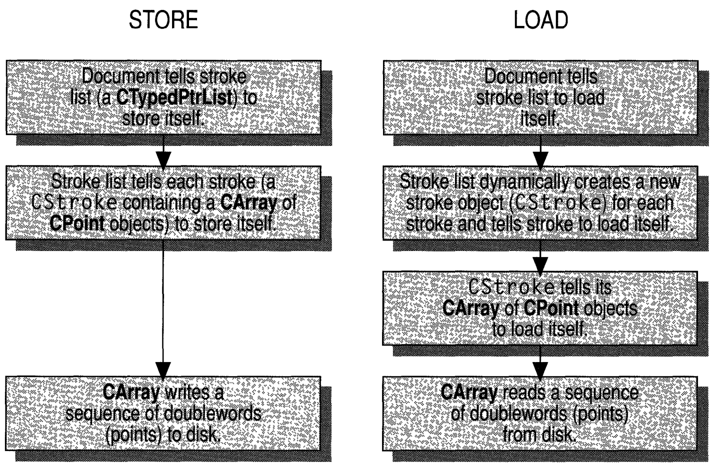

STORE
LOAD
Document tells stroke list (a CTypedPtrList) to store itself.
Stroke list tells each stroke (a CStroke containing a CArray of CPoint objects) to store itself.
CArray writes a sequence of doublewords (points) to disk.
Document tells stroke list to load itself.
Stroke list dynamically creates a new stroke object (CStroke) for each stroke and tells stroke to load itself.
CStroke tells its CArray of CPoint objects to load itself.
CArray reads a sequence of doublewords (points) from disk.

Figure 8.8 Serialization in Scribble

The CScribDoc class declaration in file SCRIBDOC.H begins with the following lines, which contain an important macro invocation needed for serialization (don't add this code):

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

AppWizard wrote this code for you.

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. If you're working along, add the marked lines to the indicated files.

▶ **To implement serialization for Scribble documents**

- Fill in the Serialize member function for class CScribDoc. The SCRIBDOC.CPP file defines a skeletal version of the function. Here's Serialize, with one marked line added to adapt the serialization mechanism to Scribble:

```
// OnNewDocument, then ...
void CScribDoc::Serialize( CArchive& ar )
{
    if ( ar.IsStoring( ) )
    {
    }
    else
    {
    }
▶   m_strokeList.Serialize( ar );

}
```

AppWizard creates the skeleton of the Serialize member function in class CScribDoc; you simply fill in the code shown. Later you'll add code in both branches of the **if** statement. AppWizard also generates comments of the form

```
// TODO: Add storing code here
```

You will typically remove these comments when the functionality has been implemented.

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 new **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.

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.

▶ **To implement serialization for stroke objects**

1. Add the **IMPLEMENT_SERIAL** macro for `CStroke`.

```
  // End of CScribDoc code in file SCRIBDOC.CPP
▶ IMPLEMENT_SERIAL( CStroke, CObject, 1 )
```

   If you're working along, add the **IMPLEMENT_SERIAL** macro for `CStroke` as shown. The third argument is the schema number, set to 1 for Scribble step 1.

   The **IMPLEMENT_SERIAL** macro complements the **DECLARE_SERIAL** macro invoked in SCRIBDOC.H. The two macros prepare a class for serialization.

2. Add a `Serialize` override for class `CStroke` in the SCRIBDOC.CPP file. Like `CScribDoc`, `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. Here's the code for `CStroke`'s version of `Serialize`:

```
  // CStroke::CStroke, then ...
▶ 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 for storing, the stroke's pen-width value is stored in the archive and then its array of points is stored. Notice that the **CArray<CPoint, CPoint>** object `m_pointArray` can serialize itself.

   If the archive object is 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 8.7 on page 106).

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 the Windows version 3.1 operating system and 32 bits in the Windows NT operating system. 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.

In the Next Chapter

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 compile. In Chapter 9, 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 compile and test Scribble.

CHAPTER 9

Creating the View

In Chapter 8, you completed Scribble's document class. In this chapter, you'll add a view class that provides a "view on the document." Scribble's view class displays the strokes of a drawing and accepts user input from the mouse. By the end of this chapter, you can compile and run Scribble.

Among the things you'll develop in this chapter are:

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

You'll also get more hands-on experience with ClassWizard. ClassWizard lets you map Windows messages to message-handler member functions in your classes. As you'll see in Chapter 11, 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.

The code that you must add to fill out the framework in this chapter is mainly in the following files: SCRIBVW.H and SCRIBVW.CPP. You'll also add two more member functions to class `CStroke` in SCRIBDOC.CPP.

This chapter and Chapter 8 cover step 1 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 MYSCRIB subdirectory. At this point, your files should consist of the starter files you created with AppWizard in Chapter 7 and modified in Chapter 8. As you read this chapter, add all lines of code marked with the ▸ symbol in the left margin. At the end of this chapter, your files should closely resemble those in the SCRIBBLE\STEP1 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP1 subdirectory.

Instructions for compiling Scribble are given at the end of this chapter.

Views

The document object defines, stores, and manages the application's data. But all user interaction with the document 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 8, 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 9.1 illustrates the view's role in relation to the document.

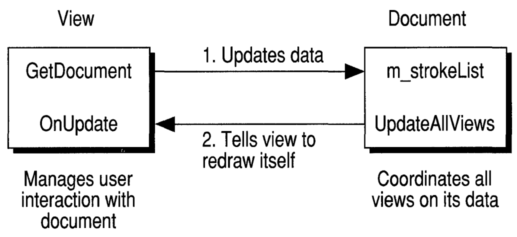

View
Document
GetDocument
OnUpdate
1. Updates data
2. Tells view to redraw itself
m_strokeList
UpdateAllViews
Manages user interaction with document
Coordinates all views on its data

Figure 9.1 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 attached to a particular document and 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 document frame window is in turn displayed inside the main frame window of the application.

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. Each Scribble document uses only one view. However, it is possible for a document to have multiple views associated with it, as in the case of splitter windows.

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 will frequently 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.

Document/View Interaction

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

Table 9.1 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 **CFormView** and **CEditView**. For more information about views, see Chapters 1, "Using the Classes to Write Applications for Windows," and 3, "Working with Frame Windows, Documents, and Views," of *Programming with the Microsoft Foundation Class Library* and class **CView** and its derived classes in the *Class Library Reference.*

Scribble's View: Class CScribView

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

AppWizard writes a skeletal `CScribView` class for you, but you need to implement the `OnDraw` member function and add a few other things. In the following procedure, as you did in Chapter 8, you'll add the indicated code to a named file. Add the lines marked with the ▸ symbol in the left margin.

▶ **To define the working data used by the view**

- Add Scribble-specific lines to class `CScribView`. File SCRIBVW.H declares class `CScribView`. Lines are added to the code generated by AppWizard to define some Scribble-specific data items:

```
  class CScribView : public CView
  {
  protected: // Create from serialization only
      CScribView( );
      DECLARE_DYNCREATE( CScribView )

  // Attributes
  public:
      CScribDoc* GetDocument( );

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

  // Operations
  .
  .
  .
  <remainder of declaration>
  .
  .
  .
  };

  #ifndef _DEBUG   // Debug version in scribvw.cpp
  inline CScribDoc* CScribView::GetDocument( )
      { return (CScribDoc*)m_pDocument; }
  #endif
```

This code declares class `CScribView`, the view on the Scribble document's data. The added lines declare two new protected member variables.

Table 9.2 describes the member variables of class `CScribView`.

Table 9.2 CScribView 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.

Table 9.3 describes the member functions of class `CScribView`.

Table 9.3 CScribView Member Functions

Member	Description
`CScribView`, `~CScribView`	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 SCRIBVW.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 14 for more information about how Scribble prints.

AppWizard creates the constructor and destructor, `GetDocument`, `AssertValid`, `Dump`, `OnPreparePrinting`, `OnBeginPrinting`, and `OnEndPrinting` for you. You won't need to alter any of these functions for the tutorial, so they are not shown in this chapter.

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 the Microsoft Foundation Class Library.*

Redrawing the View

When the view, or some part of it, must be redrawn, the framework calls your override of the `OnDraw` member function.

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

- Add `OnDraw` to file SCRIBVW.CPP, as defined below:

```
void CScribView::OnDraw( CDC* pDC )
{
    CScribDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc)
▶     // 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 to class `CStroke`.

▶ **To add drawing code for strokes**

- Add the `DrawStroke` member function to file SCRIBDOC.CPP. Its declaration is already in place in SCRIBDOC.H. `DrawStroke`, called when the view redraws itself (as described above) looks like this:

```
// CStroke::Serialize, then ...

▶ 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;
▶ }
```

`DrawStroke` is passed a pointer to an object of class **CDC**, which encapsulates a Windows device context. 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

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 ClassWizard 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. When the user subsequently releases the mouse button within the window, Windows sends the window a **WM_LBUTTONUP** message. Meanwhile, if the user moves the mouse—as in drawing—Windows sends a **WM_MOUSEMOVE** message.

How does Scribble handle these messages? They're 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 finds that it has a "handler" associated with that message name and calls the handler. The handler is a member function of class `CScribView`.

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.

What do the message-handler functions do? They track mouse activity, drawing in the view. 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 to message-handler member functions of class `CScribView`.

This step will be different from the previous ones. Instead of opening a file in the text editor and adding lines of code, you'll invoke ClassWizard and use it to make the connections between Windows messages and their handler functions. ClassWizard lets you make the connections and generate the handler functions with a few clicks of the mouse. ClassWizard writes an entry in the message map for class `CScribView` and writes a default member function definition in the SCRIBVW.CPP file for the handler function. You fill in the function's code.

You'll learn much more about ClassWizard, messages, message maps, and handler functions in Chapter 11, "Binding Visual Objects to Code Using ClassWizard." For now, if you're working along, you'll invoke ClassWizard to connect the three mouse-related messages to handler names and to generate the handlers. Then you'll use the text editor to fill in the handler functions. If you're reading along, you can still try out ClassWizard on the starter code you created with AppWizard in Chapter 7. For more information about ClassWizard, see Chapter 11 in this guide and Chapter 12, "Using ClassWizard," in the *Visual C++ User's Guide*.

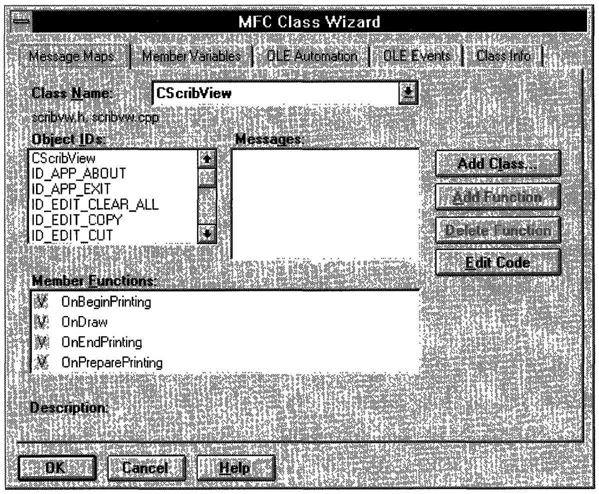
▶ **To connect the messages to Scribble's code**

1. With your Scribble project open, open ClassWizard by choosing ClassWizard from the Project menu.

   ClassWizard displays the Message Maps tab.

2. In the Class Name box, make sure class `CScribView` is selected.

   The name "CScribView" and a number of predefined command IDs appear in the Object IDs box. Figure 9.2 shows the Message Maps tab.

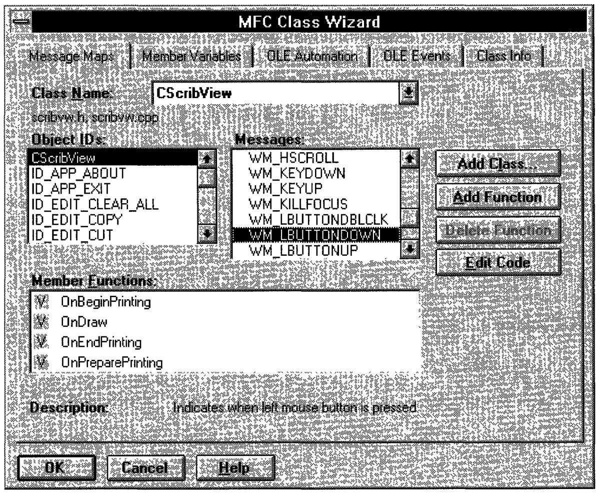


Class Name: CScribView

Figure 9.2 The ClassWizard Message Maps tab

3. In the Object IDs list box, select the name "CScribView."

   A list of overrideable functions and Windows messages that the view can receive appears in the Messages list box. Figure 9.3 shows ClassWizard with the list of virtual functions and messages.



Class Name: CScribView

Description: Indicates when left mouse button is pressed

Figure 9.3 Available Windows Messages in ClassWizard

4. In the Messages list box, scroll down past the list of overridable functions in the list of messages and select the name of a Windows message for which you want to define a handler. To begin, select **WM_LBUTTONDOWN**.

5. To define a handler function for the message, choose the Add Function button.

   The Member Functions list box now lists the member function name **OnLButtonDown** and the corresponding message-map macro name **ON_WM_LBUTTONDOWN**. A small hand-shaped icon next to the **WM_LBUTTONDOWN** message in the Messages list box shows that the connection has been made.

6. Repeat steps 4 and 5 for each additional message: first **WM_LBUTTONUP**, then **WM_MOUSEMOVE**.

After you press OK, ClassWizard will do the following things to associate each of the three messages with its handler and to greatly simplify your work:

- Added a function declaration for the handler to the `CScribView` class declaration in file SCRIBVW.H.
- Added a "message-map entry" for the message-to-handler connection in `CScribView`'s message map in file SCRIBVW.CPP.
- Added a function definition with a default body—to file SCRIBVW.CPP. For example, the default function definition for `OnLButtonDown` looks like this (don't add this code):

  ```
  void CScribView::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 `CScribView`.

You'll learn more about message maps, message-handler functions, and their uses in Chapter 11, "Binding Visual Objects to Code Using ClassWizard." For more information about these topics, see Chapter 2, "Working with Messages and Commands," in *Programming with the Microsoft Foundation Class Library.*

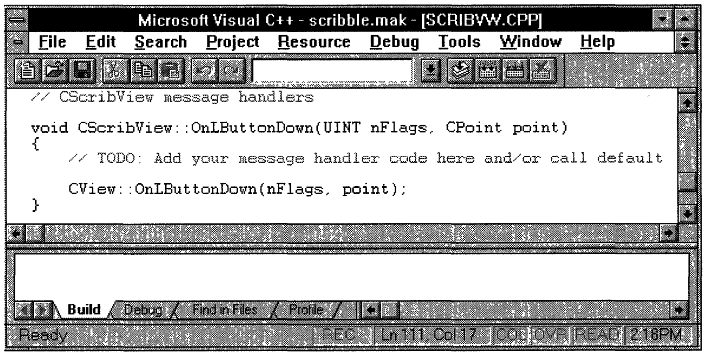
Adding the Message-Handler Functions

With the connections made, it's time to fill in the bodies of the handler functions. If you're working along, add the marked lines of code.

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

1. If you are not in ClassWizard, chose ClassWizard from the Project menu to continue.
2. Select the `OnLButtonDown` handler name in the Member Functions list box to write the handler code for the **WM_LBUTTONDOWN** message.
3. Choose the Edit Code button.

   The text editor appears, scrolled to a function definition for `OnLButtonDown`. The TODO line in its body is selected for editing. Figure 9.4 shows `OnLButtonDown` in the editor.
4. Fill in the member functions as described in the procedures that follow.


Microsoft Visual C++ - scribble.mak - [SCRIBVW.CPP]
File Edit Search Project Resource Debug Tools Window Help
Build
Debug
Find in Files
Profile
Ready
REC
Ln 111, Col 17
COL
OVR
READ
2:18PM

```
// CScribView message handlers

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

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

Figure 9.4 The Text Editor

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. If you haven't already done so, use the Edit Code button in ClassWizard or the text editor to move to the function definition for OnLButtonDown.
2. Replace the default implementation of the OnLButtonDown function body with the marked lines shown here:

```
  void CScribView::OnLButtonDown( UINT nFlags, CPoint point )
  {

▶     // When the user presses the mouse button, she may be
▶     // starting a new stroke, or selecting or de-selecting a
▶     // 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.

3. Remove the first parameter name, *nFlags*, from the declaration of OnLButtonDown in order to avoid a compiler warning that states that this parameter is not referenced.

```
▶ void CScribView::OnLButtonDown( UINT, CPoint point )
```

Once in the editor, you can complete your other message handlers or return to ClassWizard and select another function.

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 the text editor to scroll to the `OnLButtonUp` function definition in the same file.
2. Replace the default implementation of the `OnLButtonUp` function body with the marked lines shown here:

```
  void CScribView::OnLButtonUp( UINT nFlags, CPoint point )
  {
▶     // 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.
▶
▶     CScribDoc* 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;
  }
```

3. Remove the first parameter name, *nFlags*, from the declaration of `OnLButtonUp` in order to avoid a compiler warning that this parameter is not referenced.

```
▶ void CScribView::OnLButtonUp( UINT, CPoint point )
```

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. Scroll to the `OnMouseMove` function definition in the same file.
2. Replace the default implementation of the `OnMouseMove` function body with the marked lines shown here:

```
  void CScribView::OnMouseMove( UINT nFlags, CPoint point )
  {
▶     // 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;
  }
```

3. Remove the first parameter name, *nFlags*, from the declaration of `OnMouseMove` in order to avoid a compiler warning that this parameter is not referenced.

```
▶ void CScribView::OnMouseMove( UINT, CPoint point )
```

Together, these three member functions 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*.

Compile 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 the Scribble project supplied in your Visual C++ installation as described in step 2 following.

▶ **To try out your work in Chapters 8 and 9**

1. Open the SCRIBBLE.MAK project (if it's not already open) by choosing Open on the File menu and completing the dialog box. The following describes where to find the appropriate version of the project:
   - If you're building Scribble after working through the chapter, use the SCRIBBLE.MAK file in your MYSCRIB directory.

     –Or–
   - If you're simply previewing Scribble without adding code, use the SCRIBBLE.MAK file provided by Visual C++ in the SAMPLES\MFC\SCRIBBLE\STEP1 subdirectory.
2. From the Project menu, choose Build to compile and link Scribble.
3. If necessary, use the Visual C++ debugging facilities to find and correct any errors you made.

   For debugging information, see Chapter 14, "Using the Debugger," in the *Visual C++ User's Guide.*

After compiling Scribble, try out its features. 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 9.5.

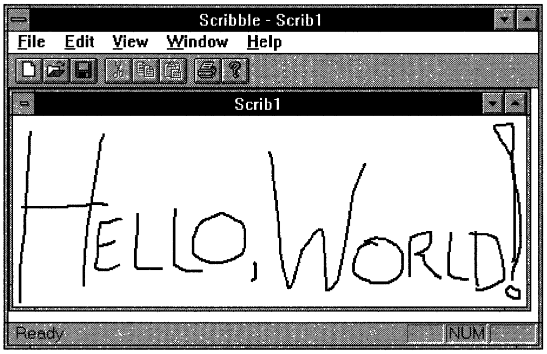


Figure 9.5 Scribble Step 1

▶ **To give Scribble a try**

1. From the Project menu, choose Execute to run Scribble.
2. Move, resize, minimize, and maximize the document window.
3. Draw "Hello, World!" (or anything) in the window. Then save the file as HELLO.SCR.
4. Try the Print Preview and Print commands on the File menu.

Note Your printout may show the strokes of your scribbling at a reduced size. This is because the current version of Scribble uses the **MM_TEXT** mapping mode instead of the **MM_LOENGLISH** mapping mode. Unlike **MM_LOENGLISH**, which Scribble will use at a later stage, **MM_TEXT** defines a pixel on the printer to be much smaller than a pixel on the screen. Printing will improve later in the tutorial.

5. Close HELLO.SCR and reopen it with the Open button on the toolbar.
6. Create a new document with the New button on the toolbar and draw in the new document. (Save the new document if you like.)
7. Exit Scribble.

Note During drawing, Scribble samples points as fast as it can (as soon as the mouse moves). When Scribble's view redraws a stroke, the playback represents approximately the speed at which the stroke was originally drawn.

This concludes your quick introduction to Scribble. You've seen how to implement the document with serialization and the view with message handling. In the next chapter, 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 10

Constructing the User Interface

Now that you've implemented enough of Scribble by hand to see how it works, it's time to explore the tools that eliminate much of the handwork.

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

- This chapter explains how to use the resource editors to visually construct Scribble's menus and toolbar.
- In Chapter 11, you will use ClassWizard and the text editor to bind menu items and toolbar buttons to commands and define message-handler functions to process the commands.
- Chapter 11 shows how to use the dialog editor and ClassWizard to create a dialog box and connect it to a menu command.

This chapter and Chapter 11 cover step 2 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 9 in your MYSCRIB directory. At this point, your files should be very similar to the files in the SCRIBBLE\STEP1 subdirectory. 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 subdirectory. You'll also use the text editor to make a small addition to MAINFRM.CPP. You can compile the new version of Scribble at the end of Chapter 11.

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

Even if you don't want to add code, however, it's a good idea to work along in this chapter to familiarize yourself with the Visual C++ programming process. You can begin by making your own copy of the SCRIBBLE\STEP1 subdirectory.

Edit Scribble's Menus

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

Thanks to AppWizard, Scribble starts out with a skeleton resource file with no effort on your part. You can look at the file SCRIBBLE.RC among the files generated in Chapter 7 to see what's there.

Default Menus

The menus AppWizard generates by default 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 the 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 used to draw subsequent strokes. They can be thick or thin. The Pen Widths command brings up a dialog box that lets the user set the thick and thin widths in pixels for subsequent drawing.

In the discussion that follows, you'll see how to create the new menu items. In the next chapter, you'll see how to use ClassWizard to connect the menus to code. And, in Chapter 12, you'll see how to create the Pen Widths dialog box and connect it to the menu command.

Adding the Menus

This section describes how to add all of Scribble's new menus and demonstrates the fundamental techniques for editing menus. For more information on using the menu editor, see Chapter 6, "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 menu item, the Clear All command, 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. From the File menu, choose Open.
2. From the File box, choose SCRIBBLE. MAK.

   The project window appears.
3. Double-click SCRIBBLE.RC.

   This launches the resource browser and automatically launches the resource compiler. Once the file has been compiled, Scribble's resource types appear in the resource browser.
4. Double-click Menu.

   Two menu IDs appear: IDR_MAINFRAME and IDR_SCRIBTYPE. IDR_MAINFRAME is the menu resource for the multiple document interface (MDI) frame window when no documents are displayed in its child windows. IDR_SCRIBTYPE is the menu resource for the frame window when a document in a child window has the focus. (Under the multiple document interface, the available menus vary depending on context.) This ID was defined for you by AppWizard. The ID name is based on the document type you chose in AppWizard.
5. Double-click IDR_SCRIBTYPE.

   The menu editor appears. You see the menus much as you would see them in the running application.
6. Click Scribble's Edit menu.

   The menu drops down as it would in the application. An empty cell outlined with a dotted line sits below the last menu item, as shown in Figure 10.1. The cell defines where you add the next menu item.

   For a taste of what you can do with menus, see the topic "Drag and Drop" on page 142.
7. Click the cell at the bottom of the Edit menu.

   The cell is highlighted with a ragged outline. This is where you'll see the menu's caption after you type it in the Properties window.

8. If the Properties window has not opened, you can:

   Double-click the cell.

   –Or–

   - Press the right mouse button while the menu editor has the focus.

     The resource object menu appears.

   - Choose Properties from the resource object menu.

     The Properties window now shows the properties of this particular menu resource. Figure 10.1 shows the menu editor and a general property page for the resource. Note the empty cells in the menu bar and Edit menu.

   –Or–

   Start typing the caption. (See the next step.)

Note In order for the Properties window to appear automatically when you open a resource, the pushpin in the upper-left corner of the Properties window must be down.

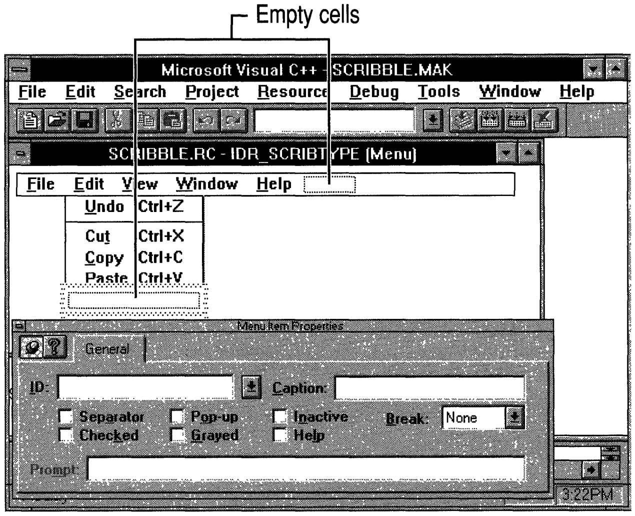


Figure 10.1 Menu Editor for IDR_SCRIBTYPE

9. Type the caption **Clear &All** in the Caption edit box of the Properties window.

   As soon as you start typing, the text you enter appears in the highlighted cell on the menu. You don't have to select the Caption edit box first.

You must type the ampersand character (**&**) in front of the letter to be used in an access key combination. As you type **&A**, for example, the letter A appears underlined in the menu.

–Or–

Type the caption and the access key combination in the cell at the bottom of the Edit menu. The Properties window appears if you haven't already opened it.

Note If you wanted to specify an accelerator or shortcut key for the menu item, you would 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.

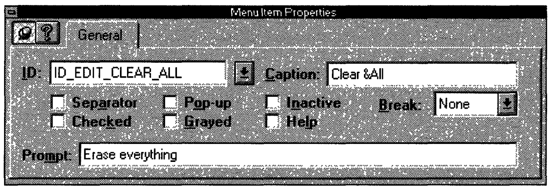
---

10. Open the ID drop-down list in the Properties window and start typing the ID for the Clear All command: **ID_EDIT_C**.

    As soon as you start typing the ID, the drop-down list box scrolls to the first ID that matches the letters you've typed so far. This behavior occurs because **ID_EDIT_CLEAR_ALL** is a command ID predefined by the class library. Visual C++ ensures that the ID you enter is unique.

    Several IDs that begin with "ID_EDIT_C" appear in the list box. Select the **ID_EDIT_CLEAR_ALL** entry.

    Figure 10.2 shows the property page after you've selected the ID.


Menu Item Properties

General

ID: ID_EDIT_CLEAR_ALL
Caption: Clear &All
☐ Separator ☐ Pop-up ☐ Inactive Break: None
☐ Checked ☐ Grayed ☐ Help
Prompt: Erase everything

Figure 10.2 Property Page with ID

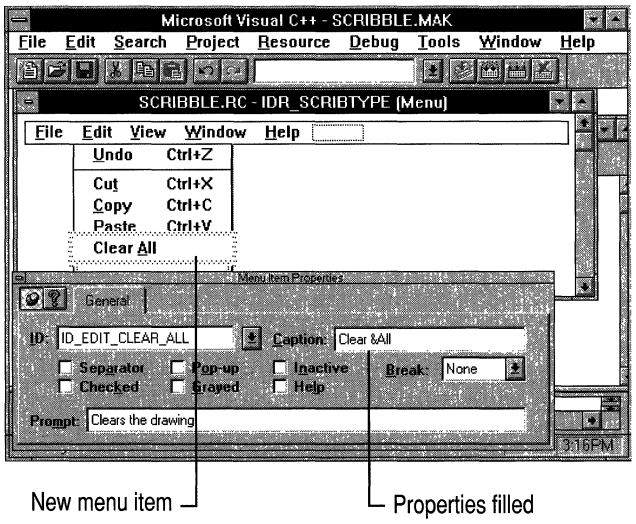
    You can define your own command IDs, of course. You'll see an example under "Add Scribble's Pen Menu" on page 140.

12. As soon as you select the predefined ID, the following string appears in the Prompt edit box: "Erase everything". Change the wording to "Clears the drawing".

    The 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.

    AppWizard predefines this text for the **ID_EDIT_CLEAR_ALL** symbol. For an 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.

That's it. You've added the Clear All command to the Edit menu. It appeared in the menu as soon as you started typing its caption. Figure 10.3 shows how the menu looks at this stage.



ID: ID_EDIT_CLEAR_ALL

Caption: Clear &All

Break: None

Prompt: Clears the drawing

Figure 10.3 Adding the Clear All Menu Item

Note You don't have to press ENTER or click any buttons to conclude your menu editing. The menu editor automatically stores the edited resource in the resource file.

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

▶ To add Scribble's Pen menu

1. With the menu-editor window still showing, click in the ragged-outlined cell at the right-hand end of the Scribble menu bar (after the Help menu).

   This cell serves the same purpose for the top-level menus as the other ragged-outlined cell does for items within a pop-up menu.

2. To position the menu entry, drag the ragged-outlined cell to the left and drop it between the Edit and View menus. (See the topic "Drag and Drop" on page 142.)
3. Type the new menu's caption, **&Pen**, in the Caption edit box of the Properties window.

   This is the only step needed to define the Pen menu as a whole. The next step is to define the menu items on the Pen menu.

   Figure 10.4 shows the new Pen menu after it has been dragged to its new location and the menu caption typed in.

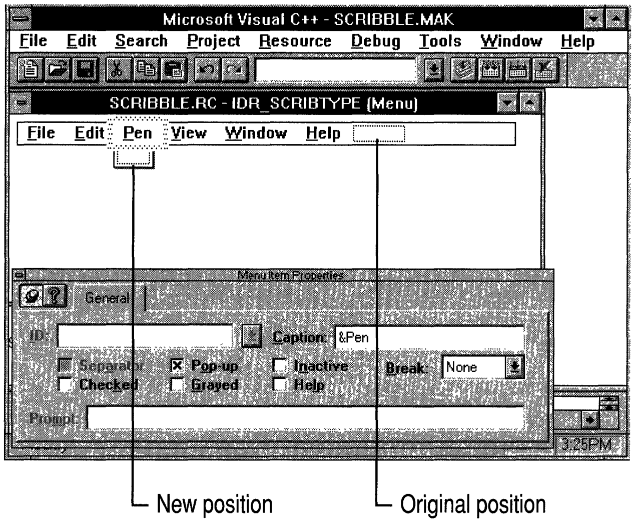

Microsoft Visual C++ - SCRIBBLE.MAK
File Edit Search Project Resource Debug Tools Window Help
SCRIBBLE.RC - IDR_SCRIBTYPE (Menu)
File Edit Pen View Window Help
Menu Item Properties
General
ID:
Caption: &Pen
Separator
Pop-up
Inactive
Break: None
Checked
Grayed
Help
Prompt:
3:25PM
New position
Original position

Figure 10.4 The Pen Menu Dragged into Position

4. Press the ENTER key to advance to the first menu item or click the ragged-outlined cell that descends beneath the word "Pen."
5. As you did for the Clear All command, type a caption for the Thick Line command in the Caption edit box: **Thick &Line.**
6. In the ID box, type **ID_PEN_THICK_OR_THIN**.
7. Type a command prompt string in the Prompt edit box:

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

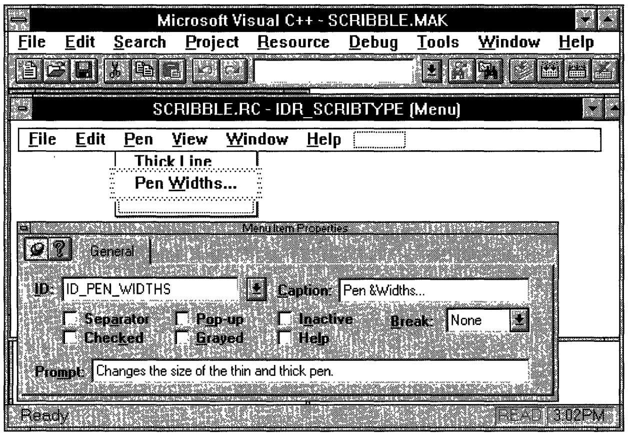
   No default string appeared because ID_PEN_THICK_OR_THIN is not a predefined command ID.

8. Click the dotted cell at the bottom of the Pen menu, below "Thick Line."
9. As you did for the Thick Line command, type a caption for the Pen Widths command: **Pen &Widths....**

   The ampersand (&) appears before the character to be used as an access key. The ellipsis (. . .) following a menu item's text lets the user know that the item brings up a dialog box.
10. In the ID box, type **ID_PEN_WIDTHS**
11. Type a command prompt string:

    ```
    Changes the size of the thin and thick pen
    ```
12. Close the menu editor.

That's all it takes to create the Pen menu. Figure 10.5 shows the completed menu as it appears in the menu editor.



Microsoft Visual C++ - SCRIBBLE.MAK

SCRIBBLE.RC - IDR_SCRIBTYPE (Menu)

ID: ID_PEN_WIDTHS   Caption: Pen &Widths...

Break: None

Prompt: Changes the size of the thin and thick pen.

3:02PM

Figure 10.5 The Completed Pen Menu

Drag and Drop

Drag and drop is a common technique in Visual C++. You'll find it in the user interfaces of many of the resource editors. For example, in Chapter 12, you'll see it used to drag various kinds of controls from a controls toolbar and drop them into a dialog box in the dialog editor. Try experimenting in the menu editor: position the mouse over an outlined box under a menu item; press the left mouse button and drag the box up or down the list and drop it where you like by releasing the mouse button. Notice that when you start to drag, an insertion point appears to orient you for dropping. You can also drag a whole menu to some other location in the menu structure, or you can drag a top-level menu to a lower level to create a hierarchical menu. If you change your mind, drag the menu back or choose Undo from the Edit menu.

Connect the Menus to Code

Typically, at this point you would open ClassWizard from the Resource menu and use it 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 11, and then return to this chapter to edit Scribble's toolbar.

Edit Scribble's Toolbar

The resource file that AppWizard creates also includes a toolbar bitmap, shown in Figure 10.6. When you build Scribble, the framework uses the toolbar bitmap to create a dockable toolbar. In this section you'll use the bitmap editor to add a new button to the bitmap for Scribble.

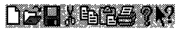


Figure 10.6 The Default Toolbar Bitmap

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 Thick Line button to Scribble's toolbar. Then, in Chapter 11, you'll use ClassWizard 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 10.7 shows Scribble as it appears with the finished toolbar. The Thick Line button is the seventh button from the left.

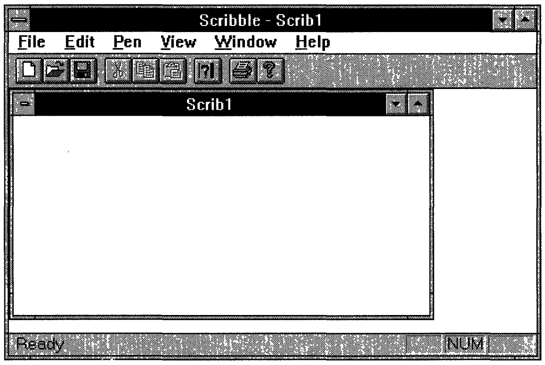


Figure 10.7 Scribble with Its Edited Toolbar

About the Toolbar

Some of the buttons on Scribble's toolbar already work, as you saw in Chapter 9 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. The print button is supported by default.

The Cut, Copy, and Paste buttons on the toolbar will not be implemented for Scribble. The Help button will be connected up in Chapter 15.

Although this chapter shows you how to add one new button, you could easily add others or delete unused buttons from the toolbar bitmap.

Add the Thick Line Button to Scribble's Toolbar Bitmap

You'll use the graphics editor for this task.

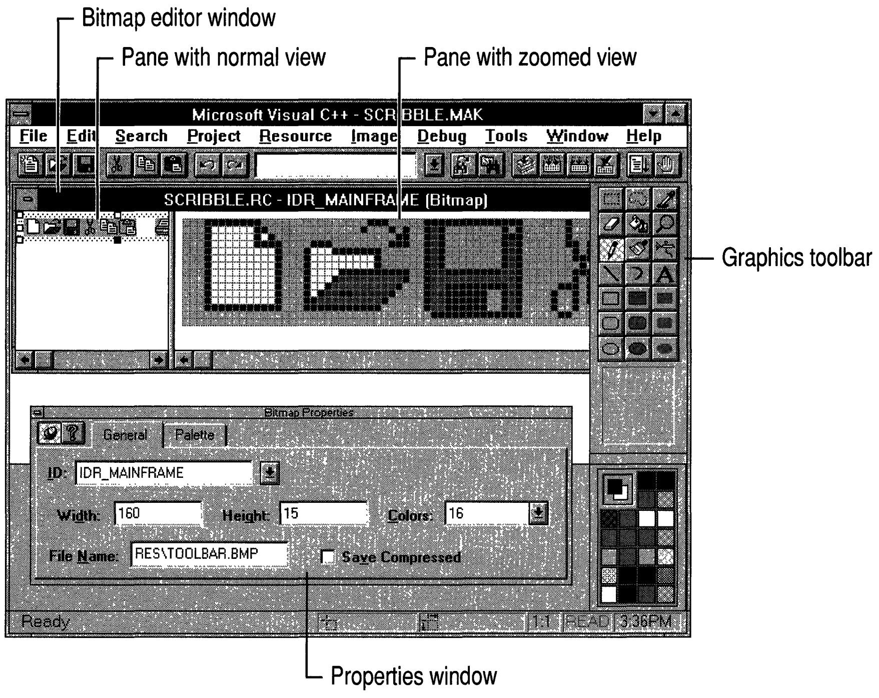
▶ **To edit Scribble's toolbar**

1. In the resource browser, double-click Bitmap.
2. Double-click IDR_MAINFRAME in the resource browser.

   An image window opens, showing the bitmap. The graphics tools open, including the graphics toolbar and the color toolbar. Figure 10.8 shows this configuration. You may see more or less of the image window than is shown in the figure, depending on your screen size.

If the graphics tools don't appear, choose Toolbars from the Tool 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 image window.



ID: IDR_MAINFRAME

Width: 160 Height: 15 Colors: 16

File Name: RES\TOOLBAR.BMP ☐ Save Compressed

Figure 10.8 The Bitmap Image Window

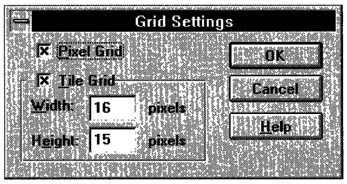
If the Properties window does not open, select IDR_MAINFRAME, press ALT+ENTER and pin down the Properties window.

3. Enlarge the visible area of the bitmap by using the graphics editor's maximize button.
4. From the Image menu, choose Grid Settings.

   The Grid Settings dialog box (Figure 10.9) appears.
5. Choose the Tile Grid checkbox and choose OK.

   Leave Pixel Grid checked as well. The size boxes show a grid size of 16 pixels by 15 pixels. This is the size of a "tile"—one of the buttons on the toolbar.

When you turn on the Tile Grid option, a thin blue rectangle (a guide) is placed around each tile in the bitmap. These guides make it easier to select and work with a tile.



Grid Settings
[✓ Pixel Grid]
[✓ Tile Grid]
Width: 16 pixels
Height: 15 pixels

Figure 10.9 The Grid Settings Dialog Box

6. Scroll the right-hand pane of the image window (which shows a zoomed image of the bitmap) until the right end of the bitmap is visible in the center of the pane.

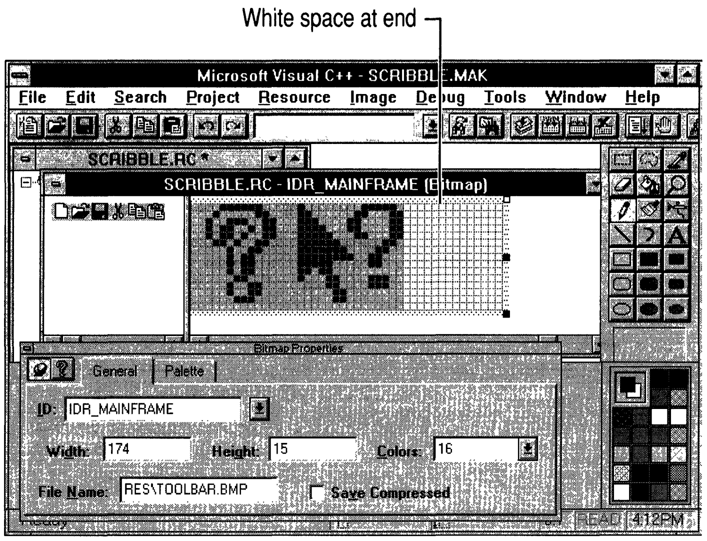
   Notice that the bitmap is outlined by a selection rectangle. In the Properties window, the Width text box shows the bitmap's current width to be 144 pixels. Beyond the end of the bitmap, there is enough space to expand the bitmap by several tiles.

7. Lengthen the bitmap by one tile: Drag the resizing handle on the right-hand end of the bitmap to the right by the width of one tile.

   The bitmap grows a full tile at a time. Notice that as you drag your bitmap, the rightmost indicator on the status bar shows new bitmap dimensions of "160 x 15." This makes the bitmap wide enough to accommodate another button.

   Beyond the current end of the bitmap, you see a white area the size of a tile at the end, overlaid with the grid. Figure 10.10 shows the bitmap as it appears after you've lengthened it, with the white space displayed at the end.

   Notice that the Width text box in the Properties window now shows the new width of the toolbar bitmap: 160 pixels.


White space at end
Microsoft Visual C++ - SCRIBBLE.MAK
File Edit Search Project Resource Image Debug Tools Window Help
SCRIBBLE.RC *
SCRIBBLE.RC - IDR_MAINFRAME (Bitmap)
Bitmap Properties
General
Palette
READ
4:12PM

ID: IDR_MAINFRAME
Width: 174 Height: 15 Colors: 16
File Name: RES\TOOLBAR.BMP
Save Compressed

Figure 10.10 The Scrolled Bitmap

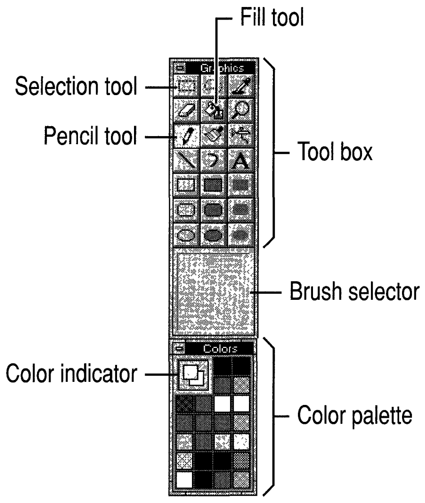


Figure 10.11 The Graphics Toolbar

8. Enclose the three rightmost button images—a printer, a question mark, and an arrow next to a question mark—with the selection rectangle.

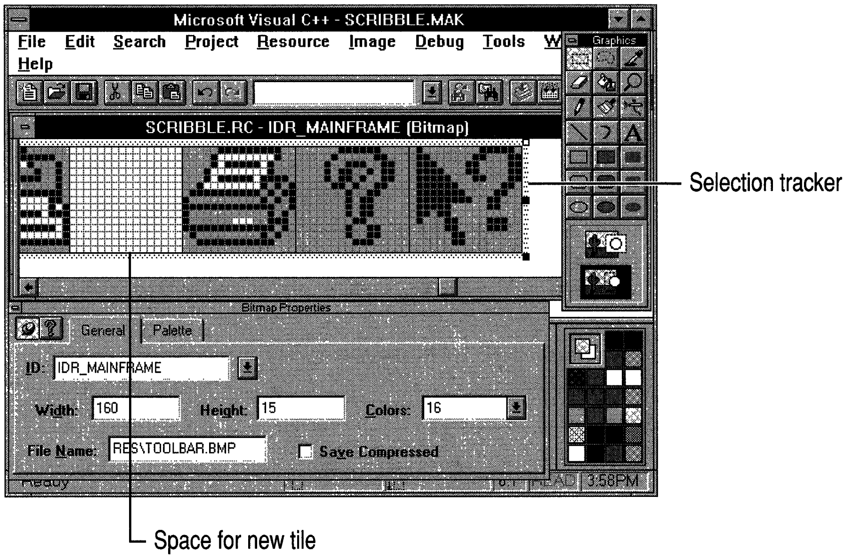
   Click the selection tool in the upper-left corner of the graphics palette.

   Figure 10.11 shows the graphics toolbar and the three tools you'll be using: the selection tool, the pencil tool, and the fill tool.

Position the mouse pointer at the upper-left corner of the printer image and drag to the lower-right corner of the question-mark image. Place the lines of the crosshairs just inside the blue guides.

9. Drag the selected images one tile to the right to open up a space for a new button.

   Figure 10.12 shows the bitmap after dragging.


Microsoft Visual C++ - SCRIBBLE.MAK
File Edit Search Project Resource Image Debug Tools W
Help
Graphics
SCRIBBLE.RC - IDR_MAINFRAME (Bitmap)
Selection tracker
Space for new tile
Bitmap Properties
General
Palette
Ready
3:58PM

ID: IDR_MAINFRAME
Width: 160 Height: 15 Colors: 16
File Name: RES\TOOLBAR.BMP
Save Compressed

Figure 10.12 The Bitmap Dragged to the Right

10. Choose the pencil tool from the graphics toolbar.
11. Draw the bitmap shown in Figure 10.13. It doesn't have to be exact.

    If you make a mistake, choose the eraser tool.

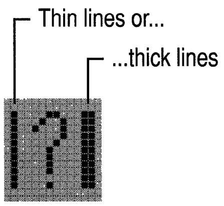


Figure 10.13 Bitmap for the Thick Line Button

12. Choose the eye dropper select color tool from the graphics toolbar.
13. Click the select color tool on the background of the last tile. The tool changes to the paint-bucket fill tool.
14. Click the fill tool in the new button image to fill its background with the selected color.

    Figure 10.14 shows the completed bitmap as it appears in Scribble.

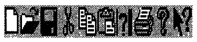


Figure 10.14 The Edited Bitmap

15. From the File menu, choose Save.

That completes the task of editing Scribble's toolbar bitmap. In order for the new button to work, it must be associated with a command ID. In this case, the Thick Line button will be bound to `ID_PEN_THICK_OR_THIN`. You defined that ID earlier in the property page for the Thick Line menu command, so Visual C++ has already written a **#define** for the ID in a file called RESOURCE.H. Your only task at this point is to associate the ID with the button. If you are working along, add lines of code marked with ▸.

Note If you had just created a new button for which there was no existing command ID, you'd use the symbol editor to define a symbol such as `ID_PEN_THICK_OR_THIN`. For information about the symbol editor, see Chapter 13, "Browsing through Symbols," in the *Visual C++ User's Guide.*

▶ **To associate the button with its command ID**

1. From the project window, open MAINFRM.CPP.
2. Scroll to the static array called "buttons" and add the marked lines, which map the buttons to their command IDs.

```
  // Arrays of IDs used to initialize control bars

  // Toolbar buttons - IDs are command buttons
  static UINT BASED_CODE buttons[] =
  {
      // Same order as in the bitmap 'toolbar.bmp'
      ID_FILE_NEW,
      ID_FILE_OPEN,
      ID_FILE_SAVE,
          ID_SEPARATOR,
      ID_EDIT_CUT,
      ID_EDIT_COPY,
      ID_EDIT_PASTE,
          ID_SEPARATOR,
▶     ID_PEN_THICK_OR_THIN,
▶         ID_SEPARATOR,
      ID_FILE_PRINT,
      ID_APP_ABOUT,
  };
```

This array definition provides a 1-to-1 mapping based on the positions of the button tiles in the toolbar bitmap. The **ID_SEPARATOR** entries denote small amounts of extra space used to group the button tiles.

3. Save the file.

Now the Thick Line button generates precisely the same command as the Thick Line menu item. Because of this, the same handler function serves for both.

Summary

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 the *Visual C++ User's Guide.*

After editing your application's menus and toolbar, the next step is to connect them to code using ClassWizard. That step is explained in Chapter 11.

CHAPTER 11

Binding Visual Objects to Code Using ClassWizard

The version of Scribble you saw in Chapter 8 and Chapter 9 added a small amount of code to the skeleton starter files created by AppWizard—much less code than you would normally have to write to get a comparable application up and running without the framework. Considering the small amount of work, the program does quite a lot: drawing, saving, printing, even print preview.

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 9, for example, you saw that Scribble implements mouse drawing by detecting and responding to messages generated by mouse clicks and drags.

This chapter introduces a category of messages called "commands," which are messages to your application from menu items, toolbar buttons, and accelerator keys. The expanded version of Scribble developed in this chapter adds two menu items that generate commands to toggle the line thickness for drawing and to clear 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 10. Now you can use ClassWizard 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:

- Learn the fundamentals of commands and how the framework routes them to various "command target" objects in the program for handling.
- Extend your knowledge of ClassWizard, begun in Chapter 9.
- 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 since a menu item may be enabled or disabled, a button checked or unchecked.

This chapter and the previous chapter cover step 2 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 10 in your MYSCRIB directory. At this point, your files should be very similar to the files in the SCRIBBLE\STEP1 subdirectory, plus the resource changes you made in Chapter 10. As you read this chapter, perform all ClassWizard steps and add all lines of code marked in the left margin with the symbol ▸. At the end of the 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, you can print or examine the files in the SCRIBBLE\STEP2 subdirectory. However, even if you don't want to add code, it's a good idea to work along in this chapter to familiarize yourself with ClassWizard and the Visual C++ programming process. You can begin by making your own copy of the SCRIBBLE\STEP1 subdirectory.

What ClassWizard Can Do

ClassWizard is one of the Visual C++ tools that you'll use most frequently. In this chapter, you'll learn to use ClassWizard to bind commands to message-handler functions. ClassWizard can also

- Connect standard Windows messages to message-handler functions.
- Connect user-interface objects to message-handler functions.
- Create new classes, such as dialogs and extra views, documents, or frame windows.
- Add member variables to dialog classes and specify how those variables are to be initialized and validated when the dialog box is displayed.

For more information about the capabilities of ClassWizard, see Chapter 12, "Using ClassWizard," in the *Visual C++ User's Guide* and the article "ClassWizard" in Part 2 of *Programming with the Microsoft Foundation Class Library.*

When you use ClassWizard 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. ClassWizard then lets you jump directly to the text editor to fill in the function template.

Note ClassWizard automatically writes its changes to your files to disk. You need to save files explicitly only if you have edited them yourself (if, for example, you used the text editor to fill in a handler's code.

You can also use ClassWizard to edit existing message maps and message-handler functions. ClassWizard follows a conservative set of rules in writing to your files, writing only a few kinds of code to predictable places, so it's safe and easy to use.

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.

You may want to work like this: Create several menu items or other user-interface objects, call up ClassWizard and connect the first object to a handler, and jump to the text editor to write the handler; then return to ClassWizard and repeat the process for each of the other objects. You may prefer to create the user-interface objects, use ClassWizard to make all the connections, and then use the text editor to fill in all the handler functions. The tools are flexible enough to support the working style you prefer.

Command Concepts and Terms

Before using ClassWizard to bind commands, you'll need to know more about the related concepts and terms. This section outlines the material, and you'll find more information about these concepts in Chapter 2, "Working with Messages and Commands," of *Programming with the Microsoft Foundation Class Library.*

- Commands, messages, and control notifications

  A command is an instruction to your program to perform a certain action. Unlike a function call, a command is a "message" that is routed to various "command target" objects, each of which has an opportunity to carry out the instruction. Examples of commands include the Open, Save, and Save As commands on the File menu.

  Important To the class library, a command is identical to its ID. A menu item, toolbar button, or dialog box control is bound to a command by giving the item the same ID.

  Commands are based on the **WM_COMMAND** Windows message. Commands can be sent to frame windows, documents, views, the application itself, and other kinds of objects. These objects are discussed as "command targets."

- User-interface objects (command generators) and updating

  Menu items, buttons, and similar elements of the user interface can cause Windows to generate commands. The framework routes commands to the other objects in your program that carry them out. Keep in mind that these user-interface objects are not C++ objects.

  You can also use commands to update the visual state of menu items and buttons—for example, enabling them if they're available to the user in a given situation and disabling them if they're unavailable. For more information, see Chapter 2, "How to Update User-Interface Objects" in *Programming with the Microsoft Foundation Class Library*.

- Command targets

  Command-target objects contain their own message maps and message-handler functions. Many objects in your program can receive and respond to commands. These objects are called "command targets" and are derived from class **CCmdTarget**. These are C++ objects.

- Command bindings

  A binding associates two things, such as a menu item and the command it invokes. Commands are assigned to the user-interface objects that generate them at one end of the command routing. This is done, for example, by using the command's ID as the ID of a menu item. At the other end of the routing, commands are mapped to the name of a message-handler function. This is done by connecting the command ID and the name of the function.

  You normally do this task with ClassWizard. Usually, you'll bind some commands as you construct the user interface. Later, you'll probably revisit ClassWizard as you work with code in the text editor.

  Note You can assign a command ID to more than one user-interface object. Scribble binds its Thick Line command to both a menu item and a toolbar button. The same message-handler function and message-map entry work for both sources of the command.

- Message maps

  How does a command target know it can handle a command? For that matter, how does any object know it can handle a message? The answer is the message map of the object's class.

  Each command-target object has a message map. Message maps are tables that connect commands—or Windows messages—with the names of the member functions that handle the commands. These functions are called "message handlers." When a command target object receives a message, the object's message map is used to determine which handler function to call for the message.

You don't typically have to write message-map entries manually—ClassWizard does the job for you.

- Message handlers

  Command-target classes have member functions to handle any commands to which the target can respond. These functions are called "message handlers." In this tutorial, the terms "handler," "message handler," and "message-handler function" are equivalent.

  A considerable part of writing an application is writing message-handler functions that determine how a document, view, or other class responds to a command.

- Command routing

  How does a command from a menu or other user-interface object find its handler function? Commands are routed through a standard sequence of command-target objects on the assumption that one of them has a handler for the command. This standard routing ensures that your objects receive the commands they need to handle.

  To illustrate, consider a command message from the Clear All item in Scribble's Edit menu. In Scribble, the handler function for this command happens to be a member function of class `CScribDoc`. Here's how that command reaches its handler after the user chooses the menu item:

  1. The main frame receives the command message first.
  2. In the case of an MDI frame window, the main frame gives the currently active MDI child window a chance to handle the command.

     (In the case of an SDI window, the main frame gives the child window, the currently active window in an MDI frame window, a chance to handle the command.)
  3. Because of the standard routing, the frame window now gives its view a chance to handle the command before checking its own message map.
  4. Unlike the frame, the view checks its own message map first.
  5. Finding no handler, the view next routes the command to its associated document.
  6. The document checks its message map and, in this case, does find a handler, which gets called—and the routing stops.

If the document did not have a handler, it would route the command next to its document template. Then the command would return to the view and then to the frame window. Finally, the child window would check its message map. If that check failed as well, the command would be routed back to the MDI frame and then to the application object—the ultimate destination of unhandled commands. For more information on command handling, see "Working with Messages and Commands," Chapter 2 in *Programming with the Microsoft Foundation Class Library.*

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 ClassWizard. (The Pen Widths command is bound 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's a problem 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 `CScribDoc` 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 a later chapter, 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 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.

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, one of the frame windows, or 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

As discussed in the previous section, Scribble's Clear All command is bound to the document class. If you're working along, use the following procedure:

▶ **To bind Scribble's Clear All command**

1. From the Project menu, choose ClassWizard.
2. Select the Message Maps tab.
3. From the Class Name box, select `CScribDoc`.

   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 `CScribDoc`.

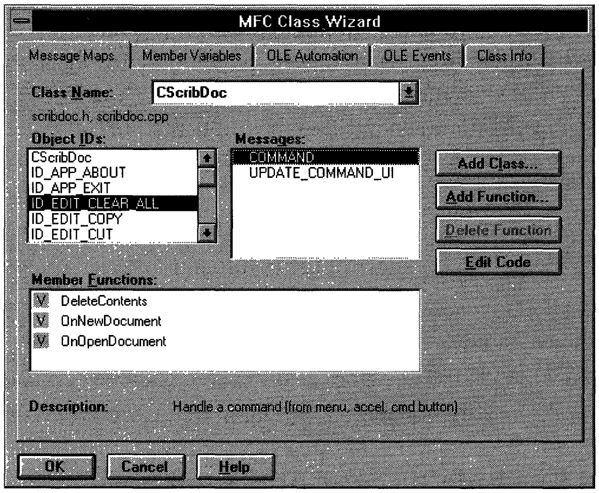
   After you select a class, the Object IDs list box shows all the visual objects managed by the class—the available items that can be mapped to functions. These might include controls (for a dialog resource) and commands from menus and accelerator tables. The list may also include class names.
4. In the Object IDs box, select **ID_EDIT_CLEAR_ALL**.

   You see **COMMAND** and **UPDATE_COMMAND_UI** in the Messages list box. For commands, these are always the choices you see. 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.

5. In the Messages box, select **COMMAND**.

Later in the chapter, you'll see how **UPDATE_COMMAND_UI** is used.

Figure 11.1 shows the selections from steps 3, 4, and 5.



Class Name: CScribDoc
scribdoc.h, scribdoc.cpp
Object IDs: CScribDoc, ID_APP_ABOUT, ID_APP_EXIT, ID_EDIT_CLEAR_ALL, ID_EDIT_COPY, ID_EDIT_CUT
Messages: COMMAND, UPDATE_COMMAND_UI
Member Functions: V DeleteContents, V OnNewDocument, V OnOpenDocument
Description: Handle a command (from menu, accel, cmd button)

Figure 11.1 Clear All in ClassWizard

6. Choose the Add Function button.

   This brings up a dialog box with a proposed name for the new handler function.

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

   Although you could change that name, don't. The name fits the handler's functionality and its connection to the menu item very well. The proposed name is synthesized from the command name and message type.

   Several things are added to your source files when you add a member function:

   - A message-map entry is added to the class's message map (in the .CPP file for the class).
   - A member function declaration is added to the class declaration (in the .H file for the class).
   - An empty member function definition is added to the .CPP file.

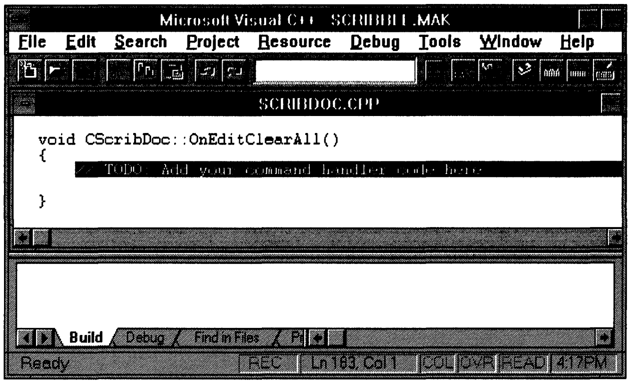
   These changes are made to your source files after you finish editing the class.

   After you add the function, its name appears in the Member Functions box beside the ID to which it maps.

8. On the Message Maps tab, make sure that the new handler's name, `OnEditClearAll`, is selected in the Member Functions box.

9. Choose the Edit Code button.

   The text editor opens SCRIBDOC.CPP and displays the skeleton code for `OnEditClearAll`. Figure 11.2 shows the skeleton code.


Microsoft Visual C++ - SCRIBBLE.MAK
File Edit Search Project Resource Debug Tools Window Help
SCRIBDOC.CPP
Build
Debug
Find in Files
Ready
REC
Ln 183, Col 1
COL
OVR
READ
4:17PM

```
void CScribDoc::OnEditClearAll()
{
    // TODO: Add your command handler code here
}
```

Figure 11.2 The OnEditClearAll Function Template

10. Add the lines marked with (▸) to fill in the `OnEditClearAll` message-handler function.

```
  void CScribDoc::OnEditClearAll( )
  {
▶         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.

11. From the File menu, choose Save to save changes to SCRIBDOC.CPP.

   This is the only file you changed by adding the new member function.

How do the commands work? The new 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 the Microsoft Foundation Class Library*.

When you finish adding OnEditClearAll, you're still in the text editor. To continue binding commands, choose ClassWizard again from the Project menu.

Bind Scribble's Thick Line Command

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?" on page 158.

▶ **To bind Scribble's Thick Line command**

1. If you're not in ClassWizard, open it from the Project menu.
2. Select the Message Maps tab.
3. From the Class Name box, select CScribDoc.
4. In the Object IDs box, select **ID_PEN_THICK_OR_THIN**.
5. In the Messages box, select **COMMAND**.
6. Choose the Add Function button.
7. In the Add Member Function dialog box, choose the OK button to accept the name OnPenThickOrThin.
8. On the Message Maps tab, choose the Edit Code button.
9. Add the lines marked with ▸▶ to fill in the OnPenThickOrThin message-handler function.

```
  void CScribDoc::OnPenThickOrThin( )
  {
▶         // 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 m_bThickPen later in the tutorial.)

10. Next, use the text editor to add `ReplacePen` to SCRIBDOC.CPP (`ReplacePen` is not a message handler, so you don't add it with ClassWizard):

```
// OnPenThickOrThin, then ...
▶    void CScribDoc::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.

11. Besides adding the `ReplacePen` function definition to SCRIBDOC.CPP, you must also add a function prototype to the `CScribDoc` class declaration in SCRIBDOC.H. If you're working along, you can open that file in the Project window, locate the class declaration as partially shown in the next code example, and add the marked line:

```
class CScribDoc : public CDocument
{
protected: // Create from serialization only.
...
// Attributes
...
// Operations
public:
...
// Implementation
▶    protected:
▶        void ReplacePen( );
...
};
```

12. To add this improved way of updating the pen, locate the `InitDocument` member function in SCRIBDOC.CPP and change the function to match this code:

```
void CScribDoc::InitDocument()
{
▶        ReplacePen();   // Initialize pen according to current width
}
```

To match the function as shown, besides adding the marked line you must delete the following lines (added in Chapter 9):

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

The line you added calls `ReplacePen` to set up the pen with its new width.

13. From the File menu, choose Save All to save changes to both SCRIBDOC.H and SCRIBDOC.CPP.

The Thick Line Command and the Toolbar Button

Scribble's Thick Line command is now bound to the Thick Line toolbar button as well as to the Thick Line menu item. Either user-interface object generates precisely the same command. This duplication is accomplished simply by giving the menu item (above) and the button (as in Chapter 10) the same ID as the command: **ID_PEN_THICK_OR_THIN**.

Add New Member Variables to Scribble

In addition to storing the current pen width in `m_nPenWidth`, class `CScribDoc` needs to keep track of whether the pen is currently thick or thin and how "thick" and "thin" are defined (in pixels). In Chapter 12, you will add code to allow the user to define these values with a dialog box. For now, defaults are hard coded.

▶ **To add the new data members**

1. In the Project window, double-click Dependencies.
2. Double-click SCRIBDOC.H.
3. Find the class declaration for class `CScribDoc`.

   It begins:

   ```
   class CScribDoc : public CDocument
   { ...
   ```

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

4. Add the following marked lines after the **protected** keyword and the existing `m_strokeList` and `m_nPenWidth` declarations:

```
  // Attributes:
  protected:
  UINT    m_nPenWidth;          // Current user-selected pen width
▶     BOOL    m_bThickPen;      // Thick currently selected or not
▶     UINT    m_nThinWidth;         // Current definition of thin
▶     UINT    m_nThickWidth;        // Current definition of thick
  CPen    m_penCur;             // Pen created according to
                                // user-selected pen style (width)

  public:
  // Additional code ...
```

5. Add the marked lines to `InitDocument` in SCRIBDOC.CPP:

```
  void CScribDoc::InitDocument()
  {
▶     m_bThickPen = FALSE;
▶     m_nThinWidth = 2;     // Default thin pen is 2 pixels wide
▶     m_nThickWidth = 5;    // Default thick pen is 5 pixels wide
      ReplacePen();   // Initialize pen according to current width
  }
```

   The added lines specify that the pen is initially thin and define the meanings of “thin” and “thick.”

6. Save files SCRIBDOC.H and SCRIBDOC.CPP.

Updating User-Interface Objects

When a menu drops down in your application, the user expects to see some menu items enabled (available) and others dimmed (grayed to show they’re unavailable) depending on the current context. Some menu items may have a check mark. Similarly, the user expects to see some toolbar buttons enabled and others disabled or perhaps checked (depressed). The framework provides a direct, command-based way to set the state of the menus and toolbar buttons as conditions in the program change. For an explanation of how this works, see Chapter 2, “Working with Messages and Commands,” in *Programming with the Microsoft Foundation Class Library*.

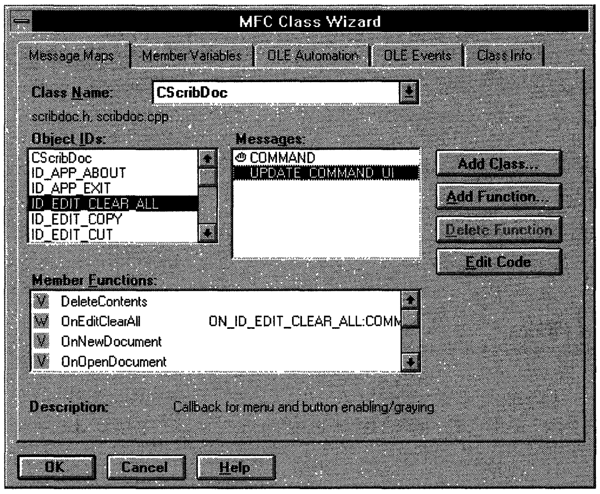
Update Scribble's Clear All Menu Item

This section presents the steps you will take to prepare the code required to update the Clear All menu item on Scribble's Edit menu. This command is handled by the document object, which has the necessary information on whether there are any strokes in the current drawing to clear. If you're working along, the following procedure guides you through the process.

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

1. From the Project menu, choose ClassWizard.
2. Select the Message Maps tab.
3. In the Class Names box, select class `CScribDoc`.
4. In the Object IDs box, select **ID_EDIT_CLEAR_ALL**.
5. In the Messages box, select **UPDATE_COMMAND_UI**.

   Figure 11.3 shows the selections made in steps 3, 4, and 5:



Class Name: CScribDoc
scribdoc.h, scribdoc.cpp

Object IDs:
CScribDoc
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_CLEAR_ALL
ID_EDIT_COPY
ID_EDIT_CUT

Messages:
COMMAND
UPDATE_COMMAND_UI

Member Functions:
V DeleteContents
W OnEditClearAll ON_ID_EDIT_CLEAR_ALL:COMM
V OnNewDocument
V OnOpenDocument

Description: Callback for menu and button enabling/graying

Figure 11.3 ClassWizard Selections for OnUpdateEditClearAll

6. Choose the Add Function button.
7. Choose the OK button in the Add Member Function dialog box to accept the name `OnUpdateEditClearAll`.
8. On the Message Maps tab, choose the Edit Code button.

9. Add the marked lines to the `OnUpdateEditClearAll` update handler function.

```
  void CScribDoc::OnUpdateEditClearAll( CCmdUI* pCmdUI )
  {
▶     // 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( ) );
  }
```

10. Save changes to SCRIBDOC.CPP.

Notice that 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 to make your update handlers fast.

When you add an update command for the ClearAll menu item, ClassWizard also writes a message-map entry in the document's message map in SCRIBDOC.CPP that looks like this:

```
BEGIN_MESSAGE_MAP( CScribDoc, CDocument )
   //{{AFX_MSG_MAP(CScribDoc)
   // Other message-map entries
   ON_UPDATE_COMMAND_UI( ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll )
   //}}AFX_MSG_MAP( )
END_MESSAGE_MAP( )
```

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

In addition, ClassWizard adds a new member function declaration for `OnUpdateEditClearAll` to the `CScribDoc` class declaration in SCRIBDOC.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, do the following procedure.

▶ **To add an update handler for the Thick Line menu**

1. From the Project menu, choose ClassWizard.
2. Select the Message Maps tab.
3. Select class `CScribDoc`.
4. In the Object IDs box, select **ID_PEN_THICK_OR_THIN**.
5. In the Messages box, select **UPDATE_COMMAND_UI**.
6. Choose the Add Function button.
7. Choose the OK button in the Add Member Function dialog box to accept the name `OnUpdatePenThickOrThin`.
8. On the Message Maps tab, choose the Edit Code button.
9. Add the marked code to the `OnUpdatePenThickOrThin` update handler function when the text editor opens.

```
  void CScribDoc::OnUpdatePenThickOrThin( CCmdUI* pCmdUI )
  {
▶     // Add check mark to Pen Thick Line menu item if the current
▶     // pen width is "thick."
▶     pCmdUI->SetCheck( m_bThickPen );
  }
```

10. Save changes to SCRIBDOC.CPP.

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 CScribDoc. 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 updating 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 file SCRIBDOC.CPP:

```
BEGIN_MESSAGE_MAP( CScribDoc, CDocument )
   //{{AFX_MSG_MAP(CScribDoc)
   // 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 declaration in SCRIBDOC.H:

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

Compile Scribble — Step 2 Version

How does Scribble behave with these new commands in place? Compile the new Step 2 version of Scribble and find out.

Run the new version of Scribble from the Project menu. Figure 11.4 shows this version of Scribble.

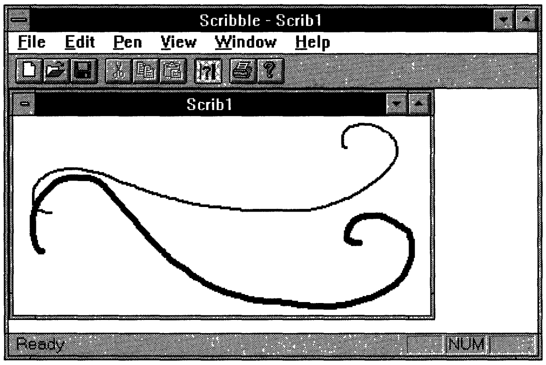


Figure 11.4 Scribble Step 2

Draw some strokes with the default thin pen. Then change the line thickness with the Thick Line command on the Pen menu and draw some new strokes. 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 now have a basic understanding of commands. In later chapters you'll build on that foundation.

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

CHAPTER 12

Adding a Dialog Box

In Chapter 10 and Chapter 11, you added new commands to Scribble in two steps: first, by using the menu editor to add new menu items; and second, by using ClassWizard to define message handlers and bind them to the commands. Recall that in Chapter 10, you added menu items for three new commands: Edit Clear All, Thick Pen, and Pen Widths. Chapter 11 discussed binding only the first two of these commands.

The reason for this omission is that 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 how thin the Thin Pen should be and how thick the Thick Pen should be. Before you can write a message handler for this 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 Chapter 10 and Chapter 11: you'll use the dialog editor to design the dialog box's appearance, and then use ClassWizard to define message handlers and bind them to the dialog box. Along the way, you'll see a feature of ClassWizard that greatly simplifies the process of gathering data from a dialog box and checking the data's validity.

This chapter describes the following topics:

- Designing a dialog box.
- Using ClassWizard to connect a class to a dialog box.
- Opening the Dialog Box from your application.

This chapter covers step 3 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 11 in your SCRIBBLE\MYSCRIB subdirectory. At this point, these files should closely resemble those in the SCRIBBLE\STEP2 subdirectory. As you read the chapter, perform all ClassWizard steps and add all the code that's marked with the symbol ▸. At the end, your files should closely resemble the files in the SCRIBBLE\STEP3 subdirectory.

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

For more information about editing dialog boxes, see Chapter 5, "Using the Dialog Editor," of the *Visual C++ User's Guide.*

Designing a Dialog Box

Figure 12.1 shows the Pen Widths dialog box that you will create.

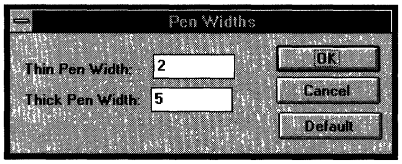


Figure 12.1 Scribble's Pen Widths Dialog Box

The Pen Width dialog box will have the following behavior: for either the Thin or Thick Pen width the user can enter any number between 1 and 20. If the user enters a value outside of this range, Scribble displays a message box stating the legal range; after dismissing the message box, the user can enter new values. To reset the pen widths to their default values, the user chooses the Default button. To use the currently displayed widths for any subsequent drawing, the user chooses the OK button. To cancel the operation, the user chooses the Cancel button.

Visual C++ provides a dialog 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), and an empty dialog box, which is the starting point for the dialog box you're designing. You select controls from the toolbar and position them on your dialog box. You can move and resize the controls directly using the mouse.

To customize the captions and IDs for the controls you've added, you open the property page for each control. In Chapter 10, you saw that menus and individual menu items have property pages; in the same way, dialog boxes and dialog controls have property pages describing their attributes.

There are three steps in designing a dialog box:

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

Create the Dialog Box

▶ **To create the Pen Widths dialog box**

1. Open SCRIBBLE.RC.
2. From the Resource menu, choose New. Then select Dialog from the list of resource types and choose OK.

   –Or–

   From the resource toolbar, choose the New Dialog button. To open the resource toolbar, choose Toolbars from the Tools menu; then select the Resource check box in the Toolbars box.

   The dialog editor window appears, displaying a dialog box that contains two buttons labeled OK and Cancel. The control toolbar also appears.
3. If the property page is not currently displayed, double-click the dialog box and then choose the push-pin button on the property page to keep it open.
4. In the ID box, change the ID to IDD_PEN_WIDTHS.
5. In the caption box, change the caption to "Pen Widths."

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

Add the Controls

▶ **To add controls to the Pen Widths dialog box**

1. From the control toolbar, drag two edit controls to the Pen Width dialog box.
   - Click on the first edit box to display its property page. Change its ID to IDC_THIN_PEN_WIDTH.
   - Bring up the property page for the second edit box. Change its ID to IDC_THICK_PEN_WIDTH.

2. From the control toolbar, drag two static text controls to contain the descriptions for the two edit controls.
   - Click the first text box to display its property page. Change the caption to read "Thin Pen Width:".
   - Bring up the property page for the other text box. Change its caption to read "Thick Pen Width:".
   - Resize each text box so that the entire caption is visible. You can do this by dragging the sizing handles on the sides of the text box or by selecting the text box and pressing F7 when the text box is selected.

   You won't have to refer 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.
   - Click the pushbutton to display its property page. Change its caption to "Default" and its ID to IDC_DEFAULT_PEN_WIDTHS.

   Note that the dialog editor has predefined the OK and Cancel buttons. If you want to look at the property pages for these buttons, click on them in turn. They have **IDOK** and **IDCANCEL**, respectively, as their IDs. Notice that the OK button has the Default Button check box chosen; this makes the OK button the default if the user immediately presses the ENTER key. Figure 12.2 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.

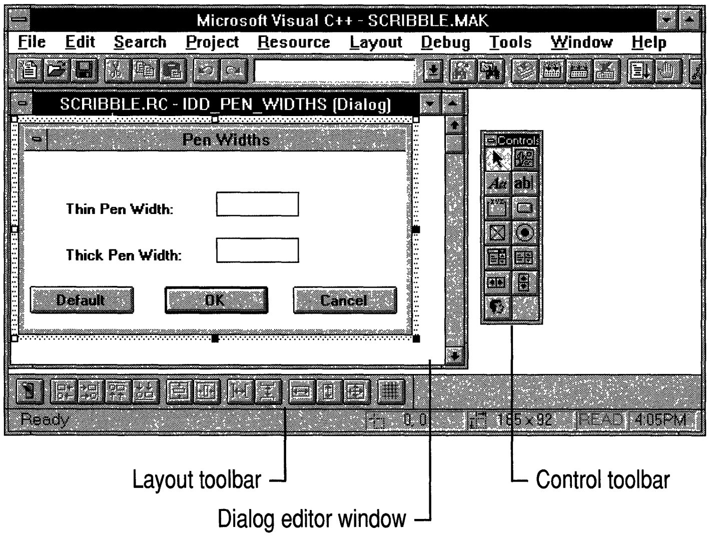


Figure 12.2 Designing the Pen Widths Dialog Box

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 if necessary.
- Arrange and align the controls

  You can improve the dialog box's appearance by using the commands on the Layout menu to align the controls, make them the same size, etc.
- 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. Click each control in tab order sequence. For more information on changing the tab order, see Chapter 5, "Using the Dialog Editor," of the *Visual C++ User's Guide*.
- 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 Resource menu. This displays the dialog box as it will appear in Scribble. Exit Test mode by choosing either the OK or Cancel button on the dialog box or by pressing the ESC key.

When you're satisfied with the way your dialog box looks, choose the Save command on the File menu. Visual C++ saves the template for your dialog box in the .RC file that you're working in, SCRIBBLE.RC in this case.

Connecting a Class to a Dialog Box

Once you've specified the appearance of your dialog box, you must specify its behavior. This requires deriving a class from **CDialog** that implements your dialog box and connecting the class to the 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 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 provides a graphical user interface that lets you do it quickly and easily. It generates a header and an implementation 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

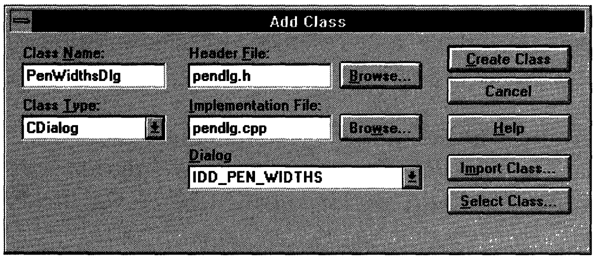
As you saw in Chapter 11, ClassWizard lets you connect a graphical object (such as a menu command or a toolbar button) to an existing class. ClassWizard also lets you declare entirely new classes in your application.

▶ **To declare a new dialog class**

1. From the Project menu, choose ClassWizard.

   The Add Class dialog box appears. ClassWizard knows that a class hasn't been defined yet, so it displays this dialog box to allow you to define one, as shown in Figure 12.3.

   You could also create the dialog class before creating the dialog resource and then use the Select Class button to connect them.



Class Name: PenWidthsDlg
Class Type: CDialog
Header File: pendlg.h
Implementation File: pendlg.cpp
Dialog: IDD_PEN_WIDTHS

Figure 12.3 The Add Class Dialog Box

2. In the Class Name box, type **CPenWidthsDlg**.

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

3. In the Header File box, change the name to "pendlg.h."

   ClassWizard offers a candidate name, which in this case was "penwidth.h", based on the class name. You can accept the suggested name or change it.

4. In the Implementation File box, change the name to "pendlg.cpp."

5. Choose the Create Class button.

   The Message Maps tab regains the focus, now displaying the name "CPenWidthsDlg" and the IDs for the controls in the Pen Widths dialog box, as shown in Figure 12.4.

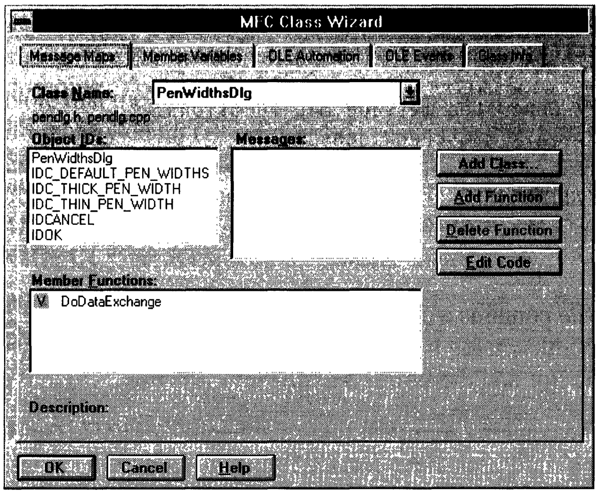


Figure 12.4 The Message Maps Tab

In previous chapters, you added code to the SCRIBDOC.H/CPP and SCRIBVW.H/CPP files, which were created by AppWizard. In this chapter you will work with two new files: PENDLG.H and PENDLG.CPP, which you specified in the Add Class dialog box. ClassWizard automatically adds these files to the project.

Header File

Here's the initial version of PENDLG.H that ClassWizard creates once you've completed the Add Class dialog box:

```
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_MSG section, containing the declarations of the message handlers; and the AFX_VIRTUAL section containing declarations of override functions. In general, you shouldn't manually edit any declarations that appear in these sections. It is good style to put any custom declarations in the appropriate group but below the //}}AFX_ line.

Implementation File

Here's the initial version of PENDLG.CPP that ClassWizard creates once you've completed the Add Class dialog box:

```
#include "stdafx.h"
#include "scribble.h"
#include "pendlg.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE 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. The `DoDataExchange` function will be described 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" on page 172. 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 the Message-Handling Functions

The **CDialog** class 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 11; both are accomplished by using ClassWizard to add an entry to a class's message map.

Once you add all the information to the Add Class dialog box in the previous section, the ClassWizard Message Maps tab is active. This is the same tab that was described in Chapter 11, but notice the following differences:

- The dialog class, not the document or view class, is the one that will handle the message. Consequently, the Class Name box displays "CPenWidthsDlg."
- 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 Message 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.

▶ **To add a message handler for the Default button**

1. In the Object IDs list box, select IDC_DEFAULT_PEN_WIDTHS as illustrated in Figure 12.4. This is the ID of the Default button. The Message list box now shows all the notification messages that a pushbutton can send, that is, **BN_CLICKED** and **BN_DOUBLECLICKED**.
2. In the Messages box, select the **BN_CLICKED** message.

   Notice the description in the Description section of the dialog box: "Indicates the user clicked a button".
3. Choose the Add Function button.

   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.
4. Choose the OK button to accept the function name offered by ClassWizard.

   The name "OnDefaultPenWidths" appears in the Member Function box and a hand-shaped icon appears next to the entry for **BN_CLICKED** in the Message box.

At this point, you could choose the Edit Code button to fill in the definition of the OnDefaultPenWidths message handler, the way you did with the message handlers for menu commands in Chapter 11. However, the purpose of this function is to manipulate member variables of the dialog class. Right now the CPenWidthsDlg class doesn't have any member variables defined; those members will be defined in the next section. You will implement OnDefaultPenWidths later in the chapter, after you've added the member variables.

Header File

Here are the changes that ClassWizard makes to PENDLG.H after you've defined the message handler (these changes are saved to the file when you close ClassWizard):

```
class CPenWidthsDlg : public CDialog
{
// Construction
public:
    CPenWidthsDlg(CWnd* pParent = NULL);

// ...

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CPenWidthsDlg)
▶   afx_msg void OnDefaultPenWidths();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Notice that ClassWizard has inserted a prototype for a member function named OnDefaultPenWidths.

Implementation File

ClassWizard makes the following changes to PENDLG.CPP after you've defined the message handler (these changes are saved to the file when you close ClassWizard):

```
  BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
      //{{AFX_MSG_MAP(CPenWidthsDlg)
▶     ON_BN_CLICKED(IDC_DEFAULT_PEN_WIDTHS, OnDefaultPenWidths)
      //}}AFX_MSG_MAP
  END_MESSAGE_MAP()

  // ...

  /////////////////////////////////////////////////////////////////////////////
  // CPenWidthsDlg message handlers

▶ void CPenWidthsDlg::OnDefaultPenWidths()
▶ {
▶     // TODO: Add your control notification handler code here
▶ }
```

ClassWizard has inserted an entry in the message map indicating that the member function `OnDefaultPenWidths` is a message handler called whenever the control **IDC_DEFAULT_PEN_WIDTHS** sends a **BN_CLICKED** message. ClassWizard has also generated an empty function definition for the message handler. You'll fill in the implementation for the function later in this chapter.

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. The Microsoft Foundation Class Library 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**

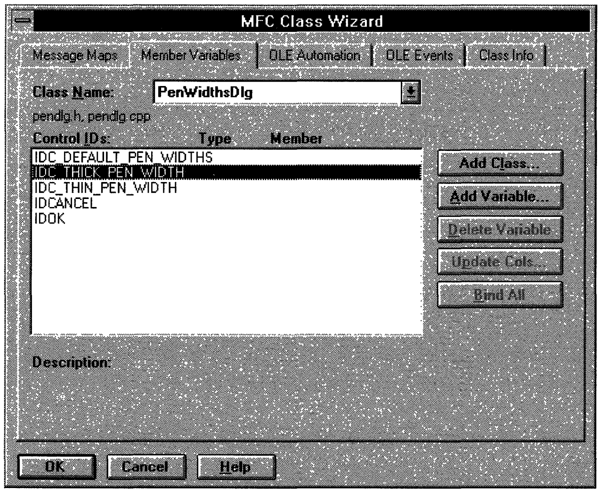
1. Choose the Member Variables tab.

   This tab, shown in Figure 12.5, contains a list box displaying the mapping between controls and member variables. 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.
2. Select **IDC_THIN_PEN_WIDTH** and then choose the Add Variable button.

   The Add Member Variable dialog box appears.
3. In the Member Name box, type **m_nThinWidth**.
4. In the Variable Type box, choose "int."
5. Choose OK to add the member variable to the class.

   Notice that the member name and type you specified now appear in the Control box and two new edit boxes appear to receive the validation parameters appropriate for an integer.
6. In the Minimum and Maximum boxes, enter "1" and "20" respectively.
7. Repeat steps 2 through 6 for the control **IDC_THICK_PEN_WIDTH**. Type **m_nThickWidth** for the member name, choose "int," and enter lower and upper limits of 1 and 20.
8. Choose OK.

You've now completed the data map connecting the Pen Widths dialog box to the `PenWidthsDlg` class.



Class Name: PenWidthsDlg
pendlg.h, pendlg.cpp

Control IDs: Type Member

IDC_DEFAULT_PEN_WIDTHS
IDC_THICK_PEN_WIDTH
IDC_THIN_PEN_WIDTH
IDCANCEL
IDOK

Description:

Figure 12.5 The Member Variables Tab

ClassWizard makes the following changes to PENDLG.H after you've mapped the controls to member variables (these changes are saved to the file after you close ClassWizard):

```
  class CPenWidthsDlg : public CDialog
  {
  // Construction
  public:
      CPenWidthsDlg(CWnd* pParent = NULL);

  // Dialog Data
      //{{AFX_DATA{CPenWidthsDlg)
      enum { IDD = IDD_PEN_WIDTHS };
▶     int      m_nThinWidth;
▶     int      m_nThickWidth;
      //}}AFX_DATA

  // ...

  };
```

ClassWizard has inserted declarations of member variables in the data map. These are the member variables you specified in the Add Member Variable dialog box.

ClassWizard makes the following changes to PENDLG.CPP after you've mapped the controls to member variables (these changes are saved to the file when you close ClassWizard):

```
  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," Chapter 12 in *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, it's time to fill in that function definition. This 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. Open ClassWizard
2. Choose the Message Maps tab.
3. Select "OnDefaultPenWidths" in the Member Functions box.
4. Choose the Edit Code button.

   The Edit Code button transfers you to the text editor, opens PENDLG.CPP, and displays the definition for `OnDefaultPenWidths`.

5. Add the marked code to the `OnDefaultPenWidths` function definition:

```
  void CPenWidthsDlg::OnDefaultPenWidths()
  {
▶     m_nThinWidth = 2;
▶     m_nThickWidth = 5;
▶     UpdateData(FALSE);  // causes DoDataExchange()
▶         // bSave=FALSE means don't save from screen, rather, write
▶         // to screen
  }
```

   The function 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.

6. Save and close the file PENDLG.CPP.

   The dialog editor window is still visible, displaying the Pen Widths dialog box you designed.

7. Click the push-pin button on the property page so that it is unpinned, then double-click the close box on the dialog editor window.

   Only the resource browser window for SCRIBBLE.RC should be visible.

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.

At the moment 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. To display the dialog box, you must call the **DoModal** member function defined by the **CDialog** class.

When you use **DoModal** to display the Pen Widths dialog box, it becomes a "modal" dialog. This means that when the dialog box is called, it takes control of the application (it puts the program in a different "mode"). The user can do no other work in the application while the dialog box is displayed and must dismiss the dialog box, typically by choosing the OK or Cancel button, to continue with the application. 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.

Now you can write a message handler for the Pen Widths command. Which class should get the handler? Recall that in Chapter 11 you added declarations for the `m_nThickWidth` and `m_nThinWidth` member variables to the `CScribDoc` 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.

▶ **To bind the Pen Widths command**

1. From the Project menu, choose ClassWizard.

   The Message Maps tab appears.
2. In the Class Name box, select "CScribDoc."
3. In the Object IDs box, select the **ID_PEN_WIDTHS** command.
4. In the Messages box, select **COMMAND**.
5. Choose the Add Function button.
6. Choose the OK button to accept the candidate name "OnPenWidths."
7. Choose the Edit Code button.

   This returns you to the text editor and opens the file SCRIBDOC.CPP.

8. Make the additions indicated by the marked lines to SCRIBDOC.CPP:

```
  #include "stdafx.h"
  #include "scribble.h"
  #include "scribdoc.h"

▶ #include "pendlg.h"

  // ...

  void CScribDoc::OnPenWidths()
  {
▶     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();
▶     }
  }
```

9. Save SCRIBDOC.CPP and SCRIBBLE.RC.

When modifying SCRIBDOC.CPP, it's necessary to include PENDLG.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.

Compile 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 compile Scribble**

- From the Project menu, choose Build scribble.exe.

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 12.6 illustrates the third version of Scribble with a variety of strokes drawn.

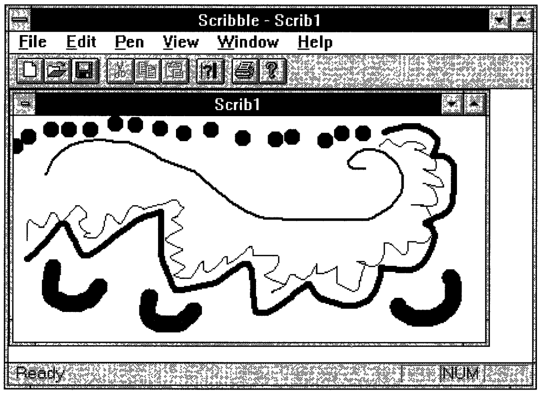


Figure 12.6 Scribble Version 3

Exit Scribble.

This completes step 3 in the tutorial.

In the next chapter, you'll implement the updating of multiple views, scrolling, and splitter windows.

CHAPTER 13

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 12 in your SCRIBBLE\MYSCRIB subdirectory. At this point, these files should closely resemble those in the SCRIBBLE\STEP3 subdirectory. As you read the chapter, add all the code marked with the symbol ▸. At the end, 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.

Updating Multiple Views

Suppose you have a drawing open in Scribble and you choose the New Window command on the Window menu. This action opens a new document window displaying the same drawing. The document object now has two view objects connected to it. Now consider what would happen if you added some new strokes in one of the document windows. Would the new strokes appear in the other window simultaneously? No, not as Scribble is currently implemented ecause each window is unaware of what's happening in the other windows. (This is illustrated in Figure 13.1.) You would have to wait until the other window is repainted (for instance, if you minimized and then restored it). Then its `OnDraw` function would display the drawing again, including the new strokes.

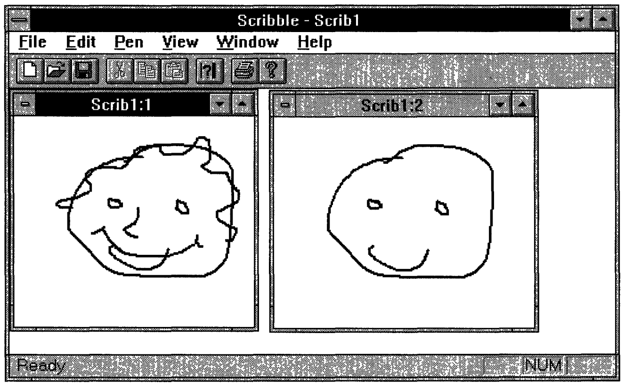


Figure 13.1 Multiple Views on a Document Without Updating

How can you ensure that all the views attached to a document reflect changes to the document as soon as they are made? Each view must notify the other views whenever it has modified the document. The Microsoft Foundation Class Library (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.

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.

▶ To define bounding rectangles for strokes

1. Open the file SCRIBDOC.H and add the following new member declarations to `CStroke`:

```
  class CStroke : public CObject
  {
  // ...
  // Attributes
  // ...
▶     CRect   m_rectBounding;   // smallest rect that surrounds all
▶                              // of the points in the stroke
▶ public:
▶     CRect& GetBoundingRect() { return m_rectBounding; }
  // Operations
  public:
      BOOL DrawStroke( );
▶     void FinishStroke();
  // ...
  };
```

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 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. There is also a new helper function, the `FinishStroke` member function.

2. Open SCRIBDOC.CPP and make the following modifications:

```
  // Each time we change what gets serialized, we change
  // the schema number.
▶ IMPLEMENT_SERIAL( CStroke, CObject, 2 )
  // ...
  CStroke::CStroke(UINT nPenWidth)
  {
      m_nPenWidth = nPenWidth;
▶     m_rectBounding.SetRectEmpty();
  }
  void CStroke::Serialize(CArchive& ar)
  {
      if (ar.IsStoring())
      {
▶         ar << m_rectBounding;
          ar << (WORD)m_nPenWidth;
          m_pointArray.Serialize(ar);
      }
else
    {
▶       ar >> m_rectBounding;
        WORD w;
        ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}
```

The changes shown here are needed to manage the addition of the `m_rectBounding` member variable. The first change to be made is incrementing the schema number in the **IMPLEMENT_SERIAL** macro. This is necessary because 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.

The next change initializes the bounding rectangle to an empty rectangle in the `CStroke` constructor. The changes to the `Serialize` member function store and read the `m_rectBounding` member variable.

3. Add the following function definition to the end of SCRIBDOC.CPP:

```
▶ void CStroke::FinishStroke()
▶ {
▶     // Calculate the bounding rectangle.  It's needed for smart
▶     // repainting.
▶
▶     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 each one's location; 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. An appropriate time to pass a hint is each time a stroke is completed.

▶ **To pass the hint after modifying the document**

- Open SCRIBVW.CPP and make the following modifications near the end of OnLButtonUp:

```
  void CScribView::OnLButtonUp(UINT, CPoint point)
  {
      // ...
      m_pStrokeCur->m_pointArray.Add( point );

▶     // 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);

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

The OnLButtonUp member function is called when a stroke is finished, so you should 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 CScribView class to respond to any hint it receives.

▶ **To use the hint for efficient repainting**

1. Open ClassWizard.
2. Choose the Message Maps tab.
3. Select the CScribView class.
4. Select "CScribView" in the Object IDs box.
5. Select "OnUpdate" in the Member Functions box.
6. Choose the Edit Code button.

   The Edit Code button transfers you to the text editor, opens SCRIBVW.CPP, and displays the definition for OnUpdate.

7. Add the marked code to the OnUpdate function definition:

```
  void CScribView::OnUpdate(CView*, LPARAM, CObject* pHint)
  {
▶ // 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 `CScribDoc`, 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.

8. Make the following changes to the `OnDraw` member function in SCRIBVW.CPP:

```
  void CScribView::OnDraw(CDC* pDC)
  {
      CScribDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

▶     // 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.
▶
        // 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);
▶           rectStroke = pStroke->GetBoundingRect();
▶           if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
▶               continue;
            pStroke->DrawStroke(pDC);
        }
    }
```

In this 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.

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 13.2.

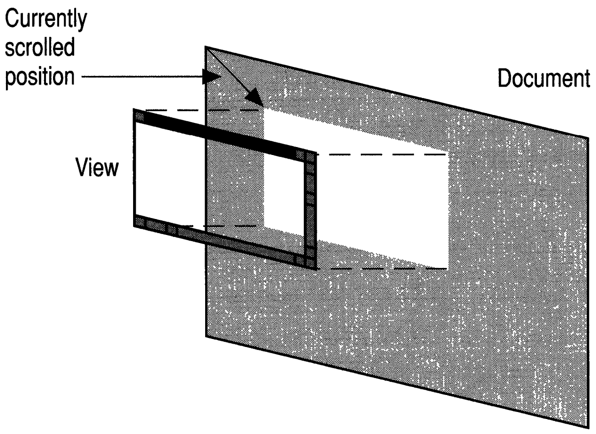


Figure 13.2 A Scrollable View on a Document

Implementing scrolling all by yourself 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**.

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**.
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 is scrolled 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 13.3 shows what Scribble looks like with scroll bars added.

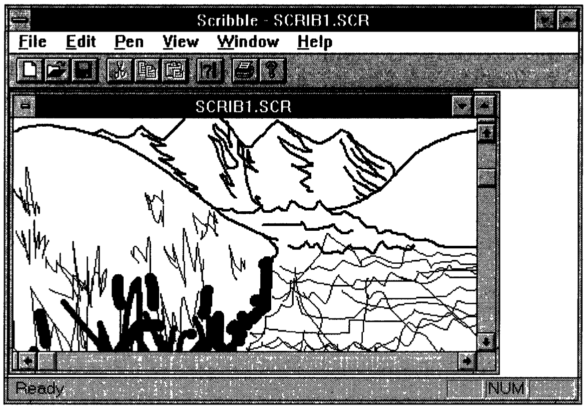


Figure 13.3 Scribble with Scrolling Support

Add Scrolling to Scribble

▶ **To add scrolling support to Scribble**

1. Open SCRIBDOC.H and make the following changes to the declaration of the `CScribDoc` class:

```
  class CScribDoc : public CDocument
  {
  // ...
  public:
      CTypedPtrList<CObList,CStroke*> m_strokeList;
▶     CPen* GetCurrentPen() { return &m_penCur; }
  // ...
▶ protected:
▶     CSize m_sizeDoc;
▶ public:
▶     CSize GetDocSize() { return m_sizeDoc; }

  // Operations
  // ...
  };
```

First define the size of Scribble documents. You can do this 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. Open SCRIBDOC.CPP and make the following changes:

```
  void CScribDoc::Serialize(CArchive& ar)
  {
      if (ar.IsStoring())
      {
▶         ar << m_sizeDoc;
      }
      else
      {
▶         ar >> m_sizeDoc;
      }
      m_strokeList.Serialize(ar);
  }

  // ...

  void CScribDoc::InitDocument()
  {
      // ...
▶     // 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 to accommodate arbitrarily large drawings.

The changes to the `Serialize` member function store and read the `m_sizeDoc` member variable.

3. Open SCRIBVW.H and make the following change to the declaration of `CScribView`:

```
▶ class CScribView : public CScrollView
  {
  // ...
  }
```

These changes set its scrolling limits according to the size of the document. By changing the base class of `CScribView` from **CView** to **CScrollView**, you give `CScribView` scrolling functionality without having to implement scrolling yourself.

In addition, the `CScribView` 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.

4. Open ClassWizard.
5. Choose the Message Maps tab.
6. Ensure that "CScribView" is selected in the Class Name and Object IDs boxes.
7. Select "OnInitialUpdate" in the Member Functions box.
8. Choose the Edit Code button.

   The Edit Code button transfers you to the text editor, opens SCRIBVW.CPP, and displays the definition for **OnInitialUpdate**.
9. Add the marked code to the **OnInitialUpdate** function definition:

```
   void CScribView::OnInitialUpdate()
   {
▶      SetScrollSizes( MM_TEXT, GetDocument()->GetDocSize() );
       CScrollView::OnInitialUpdate();
   }
```

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 14, Scribble will use the **MM_LOENGLISH** mapping mode for better printing. (For more information on mapping modes, see "Enlarge the Printed Image" in Chapter 14, 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.

10. In SCRIBVW.CPP change the following lines:

```
▶ IMPLEMENT_DYNCREATE( CScribView, CScrollView )

▶ BEGIN_MESSAGE_MAP( CScribView, CScrollView )
      //{{AFX_MSG_MAP( CScribView )
          // ...
      //}}AFX_MSG_MAP
  END_MESSAGE_MAP()
```

    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 `CScribView` all of **CScrollView**'s functionality. You also have to modify the message-map macros in the implementation file. Notice that in the message map macro, `CScribView`'s name is now followed by **CScrollView** instead of **CView**. This instructs the framework to search **CScrollView**'s message map if it can't find the message handler it needs in `CScribView`'s message map.

11. If you want to use the diagnostic features provided by MFC, change the implementations of the `Dump` and `AssertValid` member functions of `CScribView`. These functions simply call their base class versions; change them to call the **CScrollView** versions rather than the **CView** versions.

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.)

Notice that the addition of scrolling didn't require you to modify the `OnDraw` member function of `CScribView`. 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 unction 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**; in Scribble, **ScrollView**'s version of **OnPrepareDC** is called before `CScribView`'s `OnDraw` is called. 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, `CScribView` 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."

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, `CScribView` 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. Make the following modifications to the `OnLButtonDown` member function of `CScribView`:

```
void CScribView::OnLButtonDown(UINT, CPoint point)
{
▶      // 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);

       m_pStrokeCur = GetDocument()->NewStroke();
       // Add 1st 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 for the
                                //   LineTo() the next point as the user
                                //   drags the mouse.
       return;
}
```

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, 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. Make a similar modification to the `OnMouseMove` member function:

```
void CScribView::OnMouseMove(UINT, CPoint point)
{
    // ...
    if (GetCapture() != this)
        return; // If this window (view) didn't capture the mouse,
                // then the user isn't drawing in this window.

    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);

    m_pStrokeCur->m_pointArray.Add(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:

```
void CScribView::OnLButtonUp(UINT, CPoint point)
{
    // ...
    if (GetCapture() != this)
        return; // If this window (view) didn't capture the mouse,
                // then the user isn't drawing in this window.

    CScribDoc* pDoc = GetDocument();
    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);

        CPen* pOldPen = dc.SelectObject(pDoc->GetCurrentPen());
        // ...
    }
```

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. Make the following modifications to `OnUpdate`:

```
    void CScribView::OnUpdate(CView*, LPARAM, CObject* pHint)
    {
    // 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 as been added (or changed).
                // So, invalidate its rectangle.
                CStroke* pStroke = (CStroke*)pHint;
▶               CClientDC dc(this);
▶               OnPrepareDC(&dc);
                CRect rectInvalid = pStroke->GetBoundingRect();
▶               dc.LPtoDP(&rectInvalid);
                InvalidateRect(&rectInvalid);
                return;
            }
        }
        // We can't interpret the hint, so assume that anything might
        // have been updated.
        Invalidate(TRUE);
        return;
    }
```

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 your changes.

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 them 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 13.4.

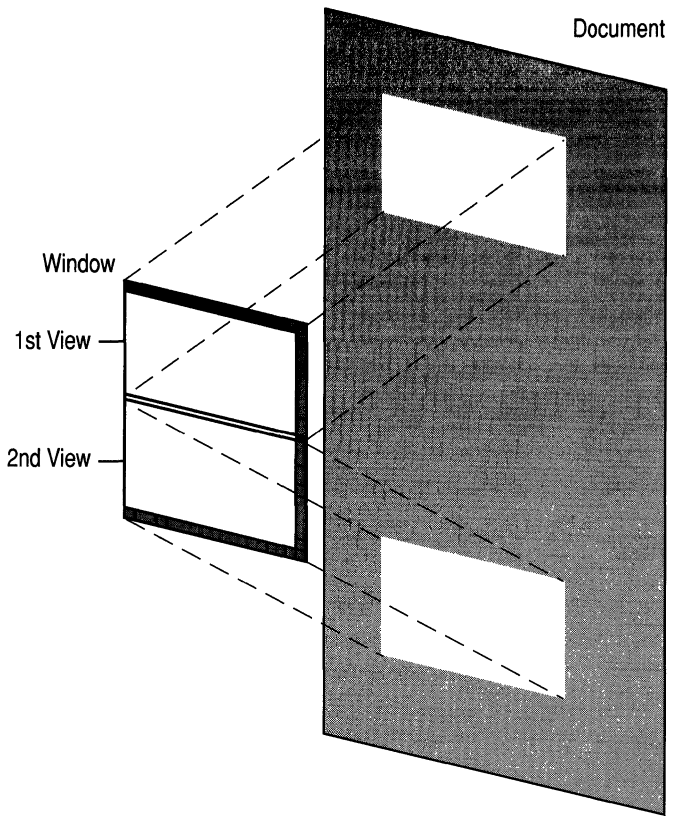


Figure 13.4 A Window with Two Views on a Document

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 13.5 shows what a Scribble window looks like when it is split into two panes.

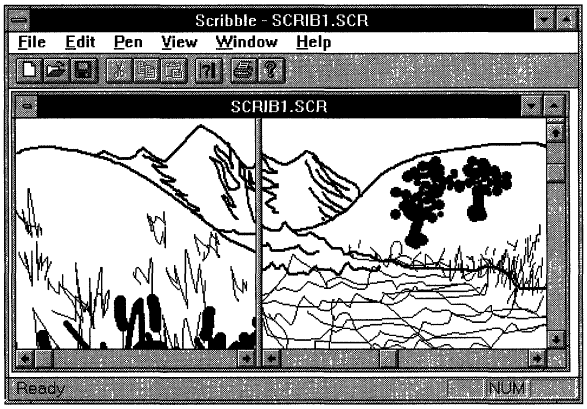


Figure 13.5 Scribble Document Window Split into Two Panes

Each pane in a splitter window represents a separate view object. In Figure 13.5, each pane is an instance of the `CScribView` 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.

The Microsoft Foundation Class Library provides splitting functionality in a class called **CSplitterWnd**. By using this class, you can support splitting in your application with very little effort.

The basic steps for adding splitter windows to your application are as follows:

1. Derive a frame window class from **CMDIChildWnd** if you are writing a Multiple Document Interface (MDI) application or **CFrameWnd** if you are writing a Single Document Interface (SDI) application. Give this class a member variable of type **CSplitterWnd**.
2. Override the **OnCreateClient** member function of your frame window class to create a **CSplitterWnd**.
3. When defining a document template, use the frame window class you derived instead of **CMDIChildWnd** or **CFrameWnd**.

There are two ways that you can add splitter window functionality to your application.

- You can choose the splitter window option in AppWizard when you create the application's skeleton. This method does all three steps for you.
- You can add this functionality manually using ClassWizard. This is the method you'll use for Scribble because it will allow you to examine in greater detail how the framework implements this feature. You'll use ClassWizard to create the class `CScribFrame`, which takes care of the first two steps.

The following describes the files that ClassWizard creates for `CScribFrame`. Similar files are created by AppWizard when you choose the splitter window option.

Header File

Here's what the header file that ClassWizard creates looks like:

```
/////////////////////////////////////////////////////////////////////////////
// CScribFrame frame with splitter

#ifndef _AFXEXT_H
#include <afxext.h>
#endif

class CScribFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE( CScribFrame )
protected:
    CScribFrame();  // protected constructor used by dynamic creation

// Attributes
protected:
    CSplitterWnd  m_wndSplitter;
public:

// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
// {{AFX_VIRTUAL (CScribFrame)
protected:
    virtual BOOL OnCreateClient( LPCREATESTRUCT lpcs,
                                 CCreateContext* pContext );
//}}AFX_VIRTUAL
▶ // Implementation
▶ public:
▶     virtual ~CScribFrame();
▶
▶     // Generated message map functions
▶     //{{AFX_MSG(CScribFrame)
▶     // NOTE - ClassWizard will add and remove member functions here.
▶     //}}AFX_MSG
▶     DECLARE_MESSAGE_MAP()
▶ };
```

`CScribFrame` assumes the role that **CMDIChildWnd** previously played in Scribble. To understand `CScribFrame`'s declaration, it's helpful to review how **CMDIChildWnd** is normally used. Until now, each time you opened a document window in Scribble, there were two objects cooperating to display the document: a **CMDIChildWnd** object, which manages the document window's frame, and a `CScribView` object, which manages the document window's client area. (These two classes were specified when AppWizard created the document template for `CScribDoc` objects.)

For Scribble to support splitting, this organization must change. Objects of three classes must cooperate to display a document: a `CScribFrame` object, which manages the document window's frame; a **CSplitterWnd** object, which manages the document window's client area; and one or more `CScribView` 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 `CScribView` 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. 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.

Now consider `CScribFrame`'s declaration. The `CScribFrame` class is derived from **CMDIChildWnd** because Scribble is an MDI application; if Scribble were an SDI application, `CScribFrame` would be derived from **CFrameWnd**. The only constructor for `CScribFrame` is a protected one because you don't need to explicitly create `CScribFrame` objects; the framework handles their creation for you.

The `CScribFrame` class defines one member variable: a **CSplitterWnd** object. This is the window that covers the frame window's client area. The class also overrides the **OnCreateClient** member function defined by **CFrameWnd** (the base class of **CMDIChildWnd**). The framework calls this function when it first creates the frame window.

Implementation File

Here's what the implementation of `CScribFrame` looks like:

```
#include "stdafx.h"
#include "scribble.h"
#include "scribfrm.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////
// CScribFrame

IMPLEMENT_DYNCREATE(CScribFrame, CMDIChildWnd)

CScribFrame::CScribFrame()
{
}

CScribFrame::~CScribFrame()
{
}

BOOL CScribFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
                            CCreateContext* pContext )
{
    return m_wndSplitter.Create( this,
        2, 2,          // TODO: adjust the number of rows, columns
        CSize( 10, 10 ),   // TODO: adjust the minimum pane size
        pContext );
}

BEGIN_MESSAGE_MAP(CScribFrame, CMDIChildWnd)
    //{{AFX_MSG_MAP(CScribFrame)
       // NOTE - ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////
// CScribFrame message handlers
```

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 `CScribFrame` window the parent of the **CSplitterWnd** object. 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.

The following section shows how to add splitter windows for Scribble.

Add Splitter Windows to Scribble

To make it easy to add splitter windows to your application, ClassWizard provides an option that automatically derives a frame window class and overrides its **OnCreateClient** member function for you. Alternatively you could add splitter windows to your application by choosing the appropriate options from AppWizard when you use it to create the skeleton files for your application. A procedure for doing this is given at the end of this section.

▶ **To add splitter windows to Scribble**

1. From the Project menu, choose ClassWizard.

   The ClassWizard dialog box appears.
2. Choose the Add Class button.

   The Add Class dialog box appears.
3. In the Class Name box, type **CScribFrame**.
4. In the Class Type box, select "splitter."
5. In the Header File box, change the default name to "scribfrm.h."
6. In the Implementation File box, change the default name to "scribfrm.cpp."
7. Choose the Create Class button to generate the new class.

   The ClassWizard dialog box regains the focus.

8. Choose the OK button to exit the ClassWizard dialog box

   As before, the new files are automatically added to the project.

9. Having defined a new frame window class, you must now use it when opening Scribble documents. Open the file SCRIBBLE.CPP and add the code indicated by the marked line:

```
  #include "stdafx.h"
  #include "scribble.h"
  #include "mainfrm.h"
▶ #include "scribfrm.h"
  #include "scribdoc.h"
  #include "scribvw.h"

  // ...

  BOOL CScribbleApp::InitInstance()
  {
      // ...

      AddDocTemplate(new CMultiDocTemplate(IDR_SCRIBTYPE,
          RUNTIME_CLASS(CScribDoc),
▶         RUNTIME_CLASS(CScribFrame),  // MDI child with splitter wnd
          RUNTIME_CLASS(CScribView)));
      // ...
  }
```

   First it's necessary to include the header file SCRIBFRM.H so you can access the declaration of the `CScribFrame` class. The major modification occurs in the `InitInstance` member function of `CScribbleApp`. This function calls **AddDocTemplate** to register the `CScribDoc` document type with the application. Recall that a document template connects a document class, a frame window class, and a view class. In previous versions of Scribble, **CMDIChildWnd** was the frame window class used for displaying `CScribDoc` objects. Now `CScribFrame`, which is derived from **CMDIChildWnd**, is the frame window class. As a result, the windows used for displaying Scribble's documents support splitting.

▶ To add splitter windows using AppWizard

1. In the AppWizard Step 4 dialog box, choose the Advanced button.
2. If you chose the Multiple Document Interface option, choose the MDI Child Frame tab.

   -Or-

   If you chose the Single Document Interface option, choose the Main Frame tab.

3. Select the Use Splitter Window option.
4. Finish choosing your options in AppWizard, and choose OK in the New Application dialog box.

For more information on **CSplitterWnd**, see the *Class Library Reference*.

Compile Scribble – Step 4 Version

How does Scribble behave with these new enhancements? Compile the new version and find out.

▶ **To compile Scribble**

- From the Project 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. Click the split box 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 13.6 shows this version of Scribble.

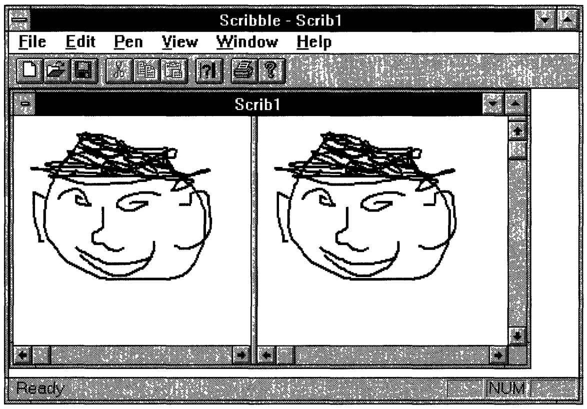


Figure 13.6 Scribble Version 4

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 14

Enhancing Printing

Scribble has supported printing and print preview since Chapter 9, 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 from Chapter 13 in your SCRIBBLE\MYSCRIB subdirectory. At this point, these files should closely resemble those in the SCRIBBLE\STEP4 subdirectory. As you read the chapter, add all the code that's marked with the symbol ▸. At the end, 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.

For more information on the framework's printing architecture, see the article "Printing" in *Programming with the Microsoft Foundation Class Library.*

Enhance Scribble's Printing

Step 5 of Scribble adds the following printing capabilities to the program:

- Enlarging the printed image to a more comfortable size.
- Paginating a Scribble document.
- Adding a page header.

The following sections describe these enhancements in detail.

Enlarge the Printed Image

Recall from Chapter 13 that when you specify a position for a GDI drawing function, you use logical coordinates. Chapter 13 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 the **MM_TEXT** mapping mode.

Since Scribble uses the **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.

In step 5, Scribble changes to the **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 the **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 13 a Scribble drawing was defined to be 800 logical units across and 900 logical units high; now a drawing is 8 inches across and 9 inches high.

Specify the Mapping Mode

You must specify the mapping mode when you call the **SetScrollSizes** member function defined by **CScrollView**. Recall from Chapter 13 that this function sets the view's scrolling limits. **SetScrollSizes** is called from the `OnInitialUpdate` member function of `CScribView`.

▶ **To specify the mapping mode**

- Open SCRIBVW.CPP and replace **MM_TEXT** wit **MM_LOENGLISH** as indicated below.

```
void CScribView::OnInitialUpdate()
{
▶·    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.

Reversing the Sign of the Y-Coordinates

Another feature of the **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.

Even though Scribble has changed the direction of the y-axis for drawing, most of the code doesn't require any modifications. This is because the **DPtoLP** function performs the conversion for you. 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. 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. Open SCRIBDOC.CPP and add the marked code to the `FinishStroke` member function of the `CStroke` class:

```
  void CStroke::FinishStroke()
  {
      // ...
      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[0];
          m_rectBounding.left     = min(m_rectBounding.left, pt.x);
          m_rectBounding.right    = max(m_rectBounding.right, pt.x);
▶         m_rectBounding.top      = max(m_rectBounding.top, pt.y);
▶         m_rectBounding.bottom   = min(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,-(int)m_nPenWidth));
      return;
  }
```

These modifications take into account the negative sign of the y coordinates.

You also must 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.

2. In SCRIBVW.CPP, make the following modifications to the `OnDraw` member function of `CScribView`:

```
  void CScribView::OnDraw(CDC* pDC)
  {
      CScribDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      // 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);
▶     pDC->LPtoDP(&rectClip);
```

```
      // Note: CScrollView::OnPaint() will have already adjusted the
      // viewport origin before calling OnDraw(), to reflect the
      // currently scrolled position.

      // 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);
          rectStroke = pStroke->GetBoundingRect();
▶         pDC->LPtoDP(&rectStroke);
          if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
              continue;
          pStroke->DrawStroke(pDC);
      }
  }
```

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.

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.

▶ To add pagination to Scribble

1. Open SCRIBVW.CPP and add the marked lines to `CScribView`'s `OnPreparePrinting` member function:

```
  BOOL CScribView::OnPreparePrinting( CPrintInfo* pInfo )
  {
▶     pInfo->SetMaxPage(2);    // the document is two pages long:
▶                              // the first page is the title page
▶                              // the second page is the drawing
      // default preparation
      return DoPreparePrinting(pInfo);
  }
```

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. Note that the function still retains a call to **DoPreparePrinting** at the end; this displays the Print dialog box and creates a device context for the printer.

2. Open SCRIBVW.H and add the marked lines to `CScribView`'s class declaration:

```
   class CScribView : public CScrollView
   {
       // ...
   //Implementation
   public:
▶      void PrintTitlePage(CDC* pDC, CPrintInfo* pInfo);
▶      void PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
▶                          CString& strHeader);
       // ...
   }
```

To perform printing, `CScribView` will override the **OnPrint** member function and defines two new helper functions: `PrintTitlePage`, which prints the title page, and `PrintPageHeader`, which prints a header on the drawing page.

3. Open ClassWizard
4. Choose the Message Maps tab.
5. Ensure that "CScribView" is selected in both the Class Name and Object IDs boxes.
6. Select "OnPrint" in the Member Functions box.
7. Choose the Edit Code button.

   The Edit Code button transfers you to the text editor, opens SCRIBVW.CPP, and displays the definition for `OnPrint`.

8. Add the following definition of the `OnPrint` member function:

```
   void CScribView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
   {
▶      if (pInfo->m_nCurPage == 1)  // page no. 1 is the title page
▶      {
▶          PrintTitlePage(pDC, pInfo);
▶          return; // nothing else to print on page 1 but the page title
▶      }
▶      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);
    }
```

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 CScribView 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.

9. In SCRIBVW.CPP, below your definition of OnPrint, define the PrintTitlePage member function as follows:

```
▶   void CScribView::PrintTitlePage(CDC* pDC, CPrintInfo* pInfo)
▶   {
▶       // 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
▶                              //  (1/100th inch)
▶       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.

Add a Page Header

As mentioned earlier, CScribView defines the PrintPageHeader function, which is called by OnPrint before the drawing itself is printed.

▶ **To add a page header to the drawing**

- In SCRIBVW.CPP, after the PrintTitlePage member function, define the PrintPageHeader member function as follows:

```
▶ void CScribView::PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
▶     CString& strHeader)
▶ {
▶     // 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 the Microsoft Foundation Class Library.*

▶ **To set the number of pages displayed in preview mode**

- In SCRIBVW.CPP, replace lines in `OnPreparePrinting` member function:

```
  BOOL CScribView::OnPreparePrinting( CPrintInfo* pInfo )
  {
      pInfo->SetMaxPage(2);    //  the document is two pages long:
                               //  the first page is the title page
                               //  the second page is the drawing
▶     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 line added here assigns 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? Compile the new version of Scribble and find out.

▶ **To compile Scribble**

- From the Project menu, choose Build.

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 14.1 shows this version of Scribble.

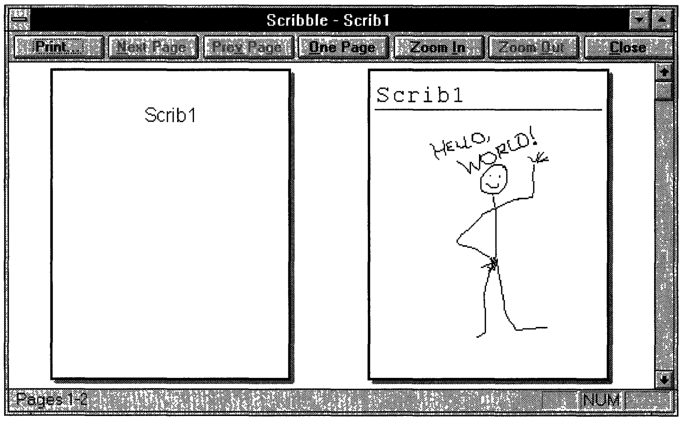


Figure 14.1 Scribble Version 5

Exit Scribble.

This completes step 5 in the tutorial. For a deeper understanding of the printing architecture provided by MFC, see the article "Printing" in *Programming with the Microsoft Foundation Class Library.*

In the next chapter, you'll add context-sensitive help to Scribble.

CHAPTER 15

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.

Note To complete this chapter, the Windows Help Compiler, which is shipped as an executable file, must be in your path.

Scribble already offers the user some help in the form of prompt strings displayed in the status bar. When the user navigates through a menu using the UP ARROW and DOWN ARROW keys, or uses the mouse to press a toolbar button, Scribble displays a brief description of the command's purpose in the status bar (if the status bar is visible). Also, 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. The framework easily supplies this level of information for commands predefined by the class library. And, as you did in Chapter 10, you can add prompts to the menu items you create by filling in a field in the menu's property page. 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 the Foundation Class Library.*

The level of help described in this chapter, however, goes much further. The user can open Windows Help for your application from the Help menu or invoke context-sensitive help by pressing the F1 key or SHIFT+F1.

This chapter explains how to implement:

- F1 help
- SHIFT+F1 help mode
- Help menu support

The next section, describes the three kinds of help listed here, and explains which functions are provided by the framework and which you must implement.

For a quick preview of how easy it is to add context-sensitive help to your application, follow the instructions described in "See Context-Sensitive Help in Action" on page 231. In that section, you'll create a new application with AppWizard, build the application, and then run it to see the help features you get without adding a single line of code.

The chapter also shows how to add an AppWizard option to your program if you didn't select the option when you originally created 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 the Microsoft Foundation Class Library.*

Note You can freely use the help files that AppWizard creates in your applications and freely ship the compiled help.

This chapter covers step 6 of Scribble. If you want to see the results of this step, you must follow the directions presented in this chapter, starting with the STEP5 source files.

Division of Labor

To support help, the framework:

- Handles F1 help.

  With an active window, dialog box, or message box, or with a menu item or toolbar button selected, the user can press the F1 key to summon specific help about the selected item.

  For menu items, help is summoned for the item currently highlighted. For toolbar buttons, the user can use the mouse to press the button and press F1 before letting the button up.

  You can define a key other than F1 for help, but it is common among applications for Windows to use F1.

- Handles SHIFT+F1 help mode.

  At any time the application is active, the user can press SHIFT+F1 to put the application into a "help mode." The cursor changes to a help cursor: an arrow beside a question mark.

While the application is in help mode, clicking any window, dialog box, message box, menu item, or toolbar button summons specific help about the item. Selecting any item for help ends help mode and displays help. Pressing the ESCAPE key or switching to another application and back also ends help mode.

The standard toolbar provided by AppWizard also has a button through which the user can invoke help mode. The graphic on the button resembles the help cursor.

You can define a key combination other than SHIFT+F1, but it is common among applications for Windows to use SHIFT+F1.

- Provides the Index and Using Help commands on the Help menu.

  The Index command causes Windows Help to display an index to the available help topics. The Using Help command causes Windows Help to display information on using Windows Help.

- Provides a starter set of files in Rich-Text Format (RTF) containing standard help topics.

  These include commands on standard menus such as File and Edit, standard information on using help, standard keyboard shortcuts, a standard help index, and more.

To take advantage of this support for help:

- Use the AppWizard Context-Sensitive Help option.
- Write your application-specific help topics in the .RTF files.

  Fill in application-specific details in these help topics, add new topics, and delete unused topics.

- Provide finer-grained context-sensitive help, if desired.

  Fine-tune 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 Microsoft Foundation Class Library in Books Online.

Implementing Context-Sensitive Help with AppWizard

Use AppWizard to enable the framework's support for context-sensitive help and the Help menu. The following sections explain how to select this support in AppWizard and what AppWizard creates as a result.

The Context-Sensitive Help Option

When you create a new application with AppWizard, be sure to select the Context-Sensitive Help option if you plan to support help.

▶ **To select context-sensitive help**

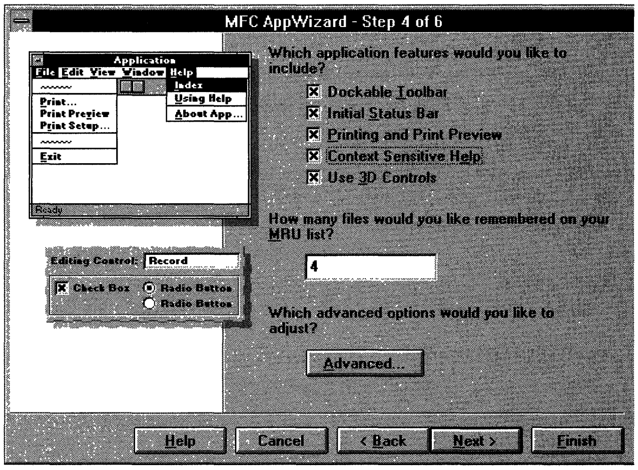
1. From the File menu, choose New.

   The New dialog box appears.
2. Select Project.

   The New Project dialog box appears.
3. In the Project Name box, type in the name of your new project and decide on the subdirectory you'd like to use for your new project. For this tutorial, use MYHELP for the project name and SAMPLES\MFC\SCRIBBLE\MYHELP for the directory.
4. Choose the options you want in AppWizard Steps 1 to 3.
5. In the Step 4 dialog box, select the Context Sensitive Help option.

   Figure 15.1 shows the Step 4 dialog box with Context Sensitive Help selected.
6. In the Steps 5 and 6 dialog boxes, select any other options you need.
7. Choose the Finish button in Step 6.

   The New Application Information dialog box appears.
8. Choose OK to create your application.


MFC AppWizard - Step 4 of 6

Which application features would you like to include?

[✓ Dockable Toolbar]
[✓ Initial Status Bar]
[✓ Printing and Print Preview]
[✓ Context Sensitive Help]
[✓ Use 3D Controls]

How many files would you like remembered on your MRU list?

4

Which advanced options would you like to adjust?

Figure 15.1 Selecting Context-Sensitive Help

When AppWizard creates your skeleton application, it adds the following items:

- 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.
- Index and Using Help items in the menu definitions.
- Status-bar command prompts for the help items. These appear when the user clicks the mouse in one of the menu commands.
- A batch file called MAKEHELP.BAT that you can use to compile your help.
- A Windows Help project file with a .HPJ extension. It's named for your project.
- One or more RTF-format files (.RTF extension) containing standard help contexts. Add application-specific help contexts to these files to customize your help. For more information, see the article "Help: Authoring Help Topics" in *Programming with the Microsoft Foundation Class Library.*
- Several bitmap files (.BMP extension) used in the help files.

You can then use the items created by AppWizard, add a few extra steps, and build your help file. These steps are described in *Programming with the Microsoft Foundation Class Library.*

Note Help project files and Windows Help tools are explained in *Programming Tools for the Microsoft Windows Operating System.*

See Context-Sensitive Help in Action

The usual way to add help to an application is to select the Context Sensitive Help option when you first run AppWizard, as just described in "Implementing Context-Sensitive Help with AppWizard."However, this was not done in Step 0 of Scribble (in order to simplify the previous chapters), so it's necessary to add the option after the fact.

However, it isn't necessary to follow the steps described in "Adding Help to Scribble After the Fact" to try out the help support provided by the framework and AppWizard. You can quickly try it out now in a newly created application.

▶ **To try out the help support**

1. Create a new application using AppWizard with the help option selected, as described in "The Context-Sensitive Help Option."

   When you run AppWizard, specify a project named MYHELP with a path of SAMPLES\MFC\SCRIBBLE\MYHELP. Select the Context Sensitive Help option. AppWizard creates help-related files for the new application.

2. Build the MYHELP application.

   It's not necessary to modify any of the code created by AppWizard. Simply build the MYHELP application that AppWizard just created.
3. Run MAKEHELP.BAT from the command line to build the .HLP file.

   As the Windows Help Compiler runs, it prints a row of dots on the screen.

Run the MYHELP application and try out various help options. Here are some suggestions for what to try:

- Choose Using Help from the Help menu. See the standard help provided by WINHELP's own help file.
- Return to the application and choose Index from the Help menu. See the standard main help topic that AppWizard has prepared. It describes the standard menus that the framework provides.
- Click the help-mode button on the 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 MYHELP's user interface.
- Using the keyboard, drop down a menu and select a menu item using the DOWN ARROW key. Then press the F1 key to get help for the selected item.

Thanks to AppWizard and the framework, you—and your users—get all of this help essentially for free.

Adding Help to Scribble After the Fact

This section explains how to add context-sensitive help at a later stage of program development. The general procedure shown here applies to any AppWizard option that wasn't chosen when the project was created.

Merging context-sensitive help support into Scribble at this late stage requires several general steps. Each step is explained in more detail below. The overall steps are:

1. Create a new MYHELP application from which to borrow code and resources for Scribble. See the following procedure "To Create a New MYHELP Application."

   The idea is to create a starter application, as in Chapter 6, that has the help-related files and code.
2. Copy resources from the MYHELP application to Scribble. See the procedure "To Copy Resources to Scribble."

3. Copy help-related code from the MYHELP application to Scribble. See the procedure "To Copy Help-Related Code to Scribble."
4. Copy help-related files from the MYHELP directory to your MYSCRIB directory. See the procedure "To Copy Help-Related Files to MYSCRIB."
5. Build the new version of Scribble and compile its help file. See the procedure "To Complete Scribble's Help."

▶ To create a new MYHELP application

- If you haven't done so already, run AppWizard to create a new MYHELP application, as described in "See Context-Sensitive Help in Action" on page 231.

  It's unnecessary to build the MYHELP application. You're about to borrow code and resources from this application for Scribble.

  The Scribble end of this procedure begins with the files from Chapter 14 (step 5) in your MYSCRIB directory. If you have not done the tutorial step in Chapter 14, you can copy all the files and subdirectories in the SAMPLES\MFC\SCRIBBLE\STEP5 subdirectory to your MYSCRIB directory.

To perform the steps in the next procedure, you'll use the menu, accelerator, and string editors, which are described in Chapters 6, 7, and 8, respectively, of the *Visual C++ User's Guide.*

▶ To copy resources to Scribble

1. Open the SCRIBBLE project in the MYSCRIB directory.

   You're about to copy menu items, accelerator keys, and status-bar prompt strings from MYHELP to MYSCRIB. 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 to another in Visual C++ into another. Note that because Scribble is an MDI application, it has two sets of menus, identified by the IDs **IDR_MAINFRAME** and **IDR_SCRIBTYPE**. You'll copy both sets.
2. Open SCRIBBLE.RC.
3. Using the File Open menu, open MYHELP.RC from the MYHELP directory.
4. Open the **IDR_MAINFRAME** menus from both resource files.

   Unmaximize the menu editor windows, and arrange them so they don't overlap.

5. Drop down both Help menus.
6. Click the separator below the Using Help item in MYHELP's Help menu. Then hold down the SHIFT key and click on the Index and Using Help items. Release the SHIFT key.

   This selects the separator and the two menu items.
7. Hold down the CTRL key, and 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.

   The menu items and the separator are copied to Scribble.
8. Close the two **IDR_MAINFRAME** menu editing windows.
9. Repeat steps 4 through 8 for the **IDR_SCRIBTYPE** menu in SCRIBBLE.RC and for the **IDR_MYHELPTYPE** menu in MYHELP.RC.
10. Use the accelerator editor to copy the accelerator keys F1 and SHIFT+F1 for the **ID_HELP** command and the **ID_CONTEXT_HELP** command, respectively.

    The copying procedure is similar to copying menus. To copy the two accelerators, hold down the SHIFT key while selecting them. Then hold down the CTRL key while dragging the accelerators to the new window.
11. Use the string editor to (a) delete the existing **AFX_IDS_IDLEMESSAGE, ID_HELP_USING,** and **ID_HELP_INDEX** strings from string segment 0 in SCRIBBLE.RC, and (b) copy the following status-bar prompt strings to MYHELP.RC: **AFX_IDS_IDLEMESSAGE**, **ID_HELP_INDEX**, **ID_CONTEXT_HELP**, **ID_HELP_USING**, and **ID_HELP**.

    The copying procedure is similar to the procedures for copying menus and accelerators. To delete a string, select it in the string editor and choose the Delete button. To copy several contiguous strings, hold the CTRL key down while selecting the strings. Then hold the CTRL key down while dragging the selected strings to the new window.

    For an application without help, AppWizard defines the default status-bar prompt to be "Ready". This is the string that is displayed in the status bar when no other command prompt is being displayed. This string is identified as **AFX_IDS_IDLEMESSAGE**.

    The other strings are command prompts for the Index and Using Help commands on the Help menu and for F1 and SHIFT+F1 help.
12. Save the MYSCRIB resource file, SCRIBBLE.RC, and close the MYHELP resource file, MYHELP.RC.

▶ **To copy help-related code to Scribble**

- Open MYHELP\MAINFRM.CPP and MYSCRIB\MAINFRM.CPP. Copy the help-related lines, marked with the ▸ symbol, from the message map in MYHELP\MAINFRM.CPP and paste them into the same position in the message map in MYSCRIB\MAINFRM.CPP. The message map looks like the following:

```
// CMainFrame

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    //{{AFX_MSG_MAP(CMyhelpApp)

        // NOTE - the ClassWizard will add and remove mapping
        // macros here.
        //     DO NOT EDIT what you see in these blocks of
        //     generated code !
    ON_WM_CREATE()
    //}}AFX_MSG_MAP


▶   // Global help commands
▶   ON_COMMAND(ID_HELP_INDEX, CMDIFrameWnd::OnHelpIndex)
▶   ON_COMMAND(ID_HELP_USING, CMDIFrameWnd::OnHelpUsing)
▶   ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
▶   ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
▶   ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpIndex)
END_MESSAGE_MAP()
```

AppWizard includes a toolbar button for help mode in the toolbar bitmap regardless of whether you choose the help option. This button did not appear on the screen when you ran previous versions of Scribble because the button was not mapped to any command in the `buttons` array defined in the MAINFRM.CPP file. The help-mode button has the rightmost position in the toolbar. Up to now there was one fewer entry in the `buttons` array than there were buttons in the toolbar bitmap. To expose the help-mode button, add the command **ID_CONTEXT_HELP** to the end of the list of commands in the `buttons` array in the MAINFRM.CPP file for Scribble.

▶ To enable the help-mode toolbar button

- Add the marked line:

```
  // toolbar buttons - IDs are command buttons
  static UINT BASED_CODE buttons[] =
  {
      // same order as in the bitmap 'toolbar.bmp'
      ID_FILE_NEW,
      ID_FILE_OPEN,
      ID_FILE_SAVE,
          ID_SEPARATOR,
      ID_EDIT_CUT,
      ID_EDIT_COPY,
      ID_EDIT_PASTE,
          ID_SEPARATOR,
      ID_PEN_THICK_OR_THIN,
          ID_SEPARATOR,
      ID_FILE_PRINT,
      ID_APP_ABOUT,
▶     ID_CONTEXT_HELP,
  };
```

▶ To copy help-related files to MYSCRIB

1. Copy the MAKEHELP.BAT and MYHELP.HPJ files from the MYHELP directory to the MYSCRIB directory.
2. In the MYSCRIB directory, rename MYHELP.HPJ as SCRIBBLE.HPJ.
3. In the copy of MAKEHELP.BAT in the MYSCRIB directory, change all occurrences of the string “myhelp” to “scribble.”
4. In SCRIBBLE.HPJ, make the following changes:
   - Under the [FILES] section, add the line

     ```
     hlp\pen.rtf
     ```

     The source and purpose of the new file PEN.RTF is explained below.
   - Under the [OPTIONS] section, change “CONTENTS=main_index” to “CONTENTS=new_index.”

     The new help topic source file, PEN.RTF, will replace the main help topic that AppWizard originally created.
   - Under the [ALIAS] section, change the string “HIDR_MAINFRAME = main_index” to “HIDR_MAINFRAME = new_index.”
   - Under the [ALIAS] section, change “HIDR_MYHELPTYPE” to “HIDR_SCRIBTYPE.”
   - Replace all occurrences of the string “myhelp” with “scribble.”

5. Create a subdirectory called HLP in your MYSCRIB directory. Copy all files in the directory MYHELP\HLP to your MYSCRIB\HLP directory. This includes several .RTF files and a number of .BMP files.
6. Copy the SCRIBBLE\STEP6\HLP\PEN.RTF file to MYSCRIB\HLP. Also, copy SCRIBBLE\STEP6\README.TXT to MYSCRIB\HLP.

   PEN.RTF and README.TXT contain help topics specific to Scribble's Pen menu. The article "Help: Authoring Help Topics" in *Programming with the Foundation Class Library* shows you some of the contents of this .RTF file and explain how you would author the help topics using a program that can edit .RTF files, such as Microsoft Word for Windows.

▶ **To complete Scribble's help**

1. Run MAKEHELP.BAT to compile your help file.
2. Compile Scribble.

Once you have successfully built Scribble and compiled its help file, run your new version of Scribble and try out its context-sensitive help.

▶ **To try out Scribble's help**

1. Press the SHIFT+F1 keys to enter help mode then click one of the items on Scribble's Pen menu.

   You'll see the custom help that has been provided in PEN.RTF.
2. Select the Index command on the Help menu to see Scribble's custom help index.

For more information on how to create a help file for an MFC application, see the Help grouping of articles in *Programming with the Microsoft Foundation Class Library*.

Conclusion

This concludes step 6. Chapter 16 describes step 7, which adds OLE server support to Scribble so that the user can embed Scribble objects into OLE container documents.

CHAPTER 16

Creating an OLE Server

An OLE visual editing server application can create Object Linking and Embedding (OLE) items that can be embedded or linked into container applications. However, some server applications only support the creation of embedded items, while others support the creation of both embedded and linked items. All server applications must be able to be started by a container 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 the Microsoft Foundation Class Library*.

Scribble Step 7 addresses two general cases for adding OLE server support to an application:

- Creating a new OLE visual editing server application from scratch
- Adding OLE visual 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 6 through 15.

As when adding OLE server support 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 a lot 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. You will need to perform the following steps.

▶ **To install and register Scribble as an OLE server application**

1. Build SCRIBBLE.EXE from SAMPLES\MFC\SCRIBBLE\STEP7 or run the prebuilt SCRIBBLE.EXE from \SAMPLES\MFC\BIN on the Visual C++ distribution CD-ROM.
2. Run Scribble briefly as a stand-alone application so that it will register itself in the system registry as an OLE server.

▶ **To install an OLE container application**

1. Build CONTAIN.EXE from SAMPLES\MFC\CONTAIN\STEP2 or run CONTAIN.EXE from \SAMPLES\MFC\BIN on the Visual C++ distribution CD-ROM.
2. From Contain's Edit menu, choose Insert New Object.
3. In the Insert Object dialog box, select Scrib Document Type from the list of Object Types to insert a Scribble object. Notice how Scribble:
   - Opens a window inside Contain for in-place editing. The window has a resize border so you can change the size of the window while visually editing the Scribble object.
   - Takes over part of Contain's menu bar and adds its own Pen menu.
   - Takes over Contain'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 built in the steps described in Chapters 6 through 15.

5. Click outside the window.

   Now the Scribble object is redrawn inside Contain's view with the help of the Scribble server.

The rest of this chapter describes how to make Scribble an OLE server.

Using AppWizard's Full Server Option

The first step for adding OLE server functionality is to run AppWizard with the full server option. If you are adding OLE support to an existing MFC application, as we are in this tutorial, you need to do the following:

- Copy your project files, in this case Scribble Step 6, to a new directory.
- Copy new files from the scratch directory to your project directory.
- Add OLE-specific server code to the application object.
- Convert the **CDocument** class to the **COleServerDoc** class.

Run AppWizard to Provide a New Skeleton

As mentioned earlier, this tutorial uses the "after-the-fact" approach of using AppWizard with exactly the same options you used in Step 1 of Scribble (see Chapter 7), to create a new skeleton with OLE features in a different "scratch" directory.

▶ **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.

   The New Project dialog box appears.
3. In the Project Name box, type **scribble.**

   The application's project file will be given this name: in this case, SCRIBBLE.MAK.

   Note If you have Visual C++ for Macintosh, please clear the Macintosh check box in the Platforms box. OLE support, as required by this tutorial, will not be generated if this check box is selected.

4. In the New Subdirectory box, delete "scribble" and type **scratch** (so you won't confuse it with the base MYSCRIB code that you will be adding to).

5. Specify the path to the project's subdirectory.

   Use the list box provided to navigate through the directories on the selected drive. As you navigate through the directory structure, the path listed in the dialog box changes to show where the named subdirectory (SCRATCH) should be placed. When the path suits you, stop navigating.

   For Scribble, navigate to SAMPLES\MFC\SCRIBBLE (relative to your Visual C++ installation). Assuming your Visual C++ installation is in directory MSVC20 on drive C, the path should look like this in the dialog box:

   ```
   c:\msvc20\samples\mfc\scribble\scratch\scribble.mak
   ```

6. Choose the Create button.

   The MFC AppWizard Step 1 dialog box appears.

7. Choose the Next button 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*.

8. In the AppWizard Step 3 dialog box, select Full-Server. Do not select Automation Support because you will not be adding automation support to Scribble.
9. Choose Next.
10. In AppWizard Step 4 dialog box, choose the Advanced button.

    The Advanced Options dialog box appears.

11. Choose the Document Template Strings tab. In the Doc Type Name box, change "Scribb" to "Scrib." In the File Extension box, type "scr."

    All the other entries in the dialog box change appropriately.

12. Choose Close.
13. Choose the Context Sensitive Help option, and choose Next.
14. In AppWizard Step 5, choose the Next button 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*.

15. In the AppWizard Step 6 dialog box, check and modify class names and filenames to make them match the original Scribble application:
    - Select the class `CScribbleDoc`, and change its name to **CScribDoc**.

      Notice that the base class is **COleServerDoc**, which reflects your Full-Server choice.

- Select the class `CScribbleView`, and change its name to **CScribView** and choose **CScrollView** in the Base Class box.
- Select the class `CScribbleSrvrItem`, and change its name to **CScribItem**, the header filename to **scribitm.h**, and the implementation filename to **scribitm.cpp**.

  Notice that **COleServerItem** is the base class, which reflects your full-server choice.

16. Choose Finish.
17. Choose Yes if you get the message box that says:

    ```
    A unique class ID already exists in the registration database for
    this document type. Use existing ID?
    ```

    This message box appears if you have already run the OLE server version of Scribble.
18. In the New Project Information dialog box, confirm the specifications and choose the OK button.
19. Now you can close the project in the SCRIBBLE\SCRATCH.

    AppWizard creates the new files.
20. Open the SCRIBBLE\MYSCRIB\SCRIBBLE.MAK project.

Copy Scribble Step 6 to a New Subdirectory

This tutorial picks up where Scribble Step 6 left off in Chapter 15. The original tutorial used the MYSCRIB subdirectory, and you will do the same here.

▶ **To copy Scribble Step 6 to a new directory**

1. Create a MYSCRIB subdirectory in SAMPLES\MFC\SCRIBBLE if you don't already have one.
2. Either use your files from Scribble Step 6, or delete all your files and copy all the files from SAMPLES\SCRIBBLE\MFC\STEP6 to the MYSCRIB subdirectory.
3. Also, copy the files from STEP6\RES to MYSCRIB\RES and from STEP6\HLP to MYSCRIB\HLP.

Transfer Scratch Files to Your MYSCRIB Project

AppWizard provides several source files for an OLE server application that you can use as-is in Scribble.

▶ **To add the new files to your MYSCRIB project**

1. Copy the following files from SCRIBBLE\SCRATCH to SCRIBBLE\MYSCRIB:
   - IPFRAME.H
   - IPFRAME.CPP
   - SCRIBITM.H
   - SCRIBITM.CPP
   - SCRIBBLE.REG
   - HLP\AFXOLESV.RTF
   - RES\ITOOLBAR.BMP
2. Make sure the MYSCRIB project is open.
3. From the Project menu, choose Files.

   The Project Files dialog box appears.
4. Add the following two files to your project:
   - IPFRAME.CPP
   - SCRIBITM.CPP
5. Choose the Close button.
6. Open SCRIBBLE.HPJ.
7. Add "hlp\afxolesv.rtf" to the [FILES] section.

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.

SCRIBITM.CPP contains the implementation of Scribble's **COleServerItem**-derived class. The **COleServerItem** object represents the Scribble document when it is embedded in a container.

SCRIBBLE.REG is a text file that can be used to register the new application with Windows. There are three ways you can register the application with Windows:

1. Run REGEDIT.EXE. From the File menu, choose Merge Registration File, and select SCRIBBLE.REG.
2. If you are redistributing your application, your installation program can spawn REGEDIT, using the /s (silent) option and specifying the .REG file as a command line parameter, as follows:

   ```
   regedit /s scribble.reg
   ```

3. Let the framework programmatically register the application for you. This third method is automatically implemented by AppWizard, as explained later. If you rely on this convenient third method, you do not need the .REG file. In this case, the .REG file is useful for informational purposes.

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.

▶ **To add AFXOLE.H to the precompiled header file**

- Add the marked line to STDAFX.H.

```
  #include <afxwin.h>      // MFC core and standard components
  #include <afxext.h>      // MFC extensions (including VB)
  #include <afxtempl.h>    // MFC templates
▶ #include <afxole.h>      // MFC OLE support
```

Add OLE Server Support to the Application Object

To add OLE server support to the application object, you need to:

- Add a **COleTemplateServer** data member to `CScribbleApp`.
- Add OLE server code to `CScribbleApp`.

▶ To add a COleTemplateServer data member to CScribbleApp

- Add the marked lines in SCRIBBLE.H.

  You will find the same code in SCRATCH\SCRIBBLE.H, which was provided by AppWizard.

```
  class CScribbleApp : public CWinApp
  {
  ...
  // Implementation
▶     COleTemplateServer m_server;
▶         // Server object for document creation
  ...
  };
```

  The **COleTemplateServer** object is used to register a server application with OLE. See how `m_server` is used in Scribble's `InitInstance`.

▶ To add OLE server code to CScribbleApp

Add OLE server code to the implementation of `CScribbleApp` in SCRIBBLE.CPP. You can take advantage of the fact that you have made very few changes to SCRIBBLE.CPP since you originally ran AppWizard in Step 1 of the Scribble tutorial. Here is a shortcut:

1. Copy SCRATCH\SCRIBBLE.CPP to MYSCRIB\SCRIBBLE.CPP.
2. Add the marked **#include** statement to the top of SCRIBBLE.CPP.

```
  ...
  #include "mainfrm.h"
▶ #include "scribfrm.h"
  #include "ipframe.h"
  ...
```

   This change also reflects the Step 4 addition of splitter window support.
3. In `CScribbleApp::InitInstance`, change:

```
pDocTemplate = new CMultiDocTemplate(
    ...
    RUNTIME_CLASS(CMDIChildWnd),
        // standard MDI child frame
        ..
```

to:

```
    pDocTemplate = new CMultiDocTemplate(
        ...
▶       RUNTIME_CLASS(CScribFrame),
▶           // MDI child frame with splitter wnd
```

This change is needed because in Step 4 of the Scribble tutorial the frame window class was changed from the standard **CMDIChildWnd** to `CScribFrame` when splitter window functionality was added.

4. Use the string editor to copy the same string resource for IDP_OLE_INIT_FAILED (used in `InitInstance`) that AppWizard provides. The string text is:

```
OLE initialization failed. Make sure that the OLE libraries are
correct version.
```

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 visual 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 `CScribDoc` from **CDocument** to **COleServerDoc**.
- Implement the document's support for embedded items.

▶ To change the base class of CScribDoc

1. In SCRIBDOC.H, change:

```
class CScribDoc : public CDocument
```

to:

```
▶ class CScribDoc : public COleServerDoc
```

2. In SCRIBDOC.CPP, replace all instances of **CDocument** with **COleServerDoc**.

   This changes the base class reference of `CScribDoc` from **CDocument** to **COleServerDoc**.

▶ **To implement the document's support for embedded items**

The **COleServerItem** object represents the Scribble document when the document is embedded in a container. To create a **COleServerItem** for a given document, **GetEmbeddedItem** is overridden 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 in the Microsoft Foundation Class Library Samples.

---

1. Open SCRIBDOC.H and add the following forward class reference for `CScribItem`:

```
  class CStroke;
▶ class CScribItem;
```

2. Declare **OnGetEmbeddedItem** as indicated by the marked line:

```
      ...
  // Implementation
      protected:
▶         COleServerItem* OnGetEmbeddedItem();
```

3. Implement `CScribDoc::OnGetEmbeddedItem` in SCRIBDOC.CPP by adding the marked lines:

```
▶ COleServerItem* CScribDoc::OnGetEmbeddedItem()
▶ {
▶     CScribItem* pItem = new CScribItem(this);
▶     ASSERT_VALID(pItem);
▶     return pItem;
▶ }
```

4. In SCRIBDOC.CPP, add the marked line.

```
  #include "scribdoc.h"
▶ #include "scribitm.h"
```

5. For convenience you provide a type-safe function to return a pointer to the specific **COleServerItem**-derived class, `CScribItem`, by adding the marked lines in SCRIBDOC.H:

```
  //Attributes
  // ...
      public:
          CSize GetDocSize() {return m_sizeDoc;}
▶         CScribItem* GetEmbeddedItem()
▶         { return (CScribItem*)COleServerDoc::GetEmbeddedItem(); }
```

Analyze OLE Server Code in InitInstance

The code highlighted below shows the OLE server code provided by AppWizard in SCRIBBLE.CPP.

```
▶ #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.
▶ static const CLSID BASED_CODE clsid =
▶ { 0x0002180f, 0x0, 0x0, { 0xC0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x46 } };

  ///////////////////////////////////////////////////////////////////////////
  // CScribbleApp initialization

  BOOL CScribbleApp::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.

      Enable3dControls();
      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_SCRIBTYPE,
            RUNTIME_CLASS(CScribDoc),
            RUNTIME_CLASS(CScribFrame)  // MDI child frame with splitter
            RUNTIME_CLASS(CScribView));
▶       pDocTemplate->SetServerInfo(
▶           IDR_SCRIBTYPE_SRVR_EMB, IDR_SCRIBTYPE_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 DDE Execute open
        EnableShellOpen();
        RegisterShellFileTypes();

▶       // Parse the command line to see if launched as OLE server
▶       if (RunEmbedded() || RunAutomated())
▶       {
▶           // 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);

      // simple command line parsing
      if (m_lpCmdLine[0] == '\0')
      {
          // create a new (empty) document
          OnFileNew();
      }
      else
      {
          // open an existing document
          OpenDocumentFile(m_lpCmdLine);
      }

      m_pMainWnd->DragAcceptFiles();
      // 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, which 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_SCRIBTYPE_SRVR_EMB, IDR_SCRIBTYPE_SRVR_IP,
    RUNTIME_CLASS(CInPlaceFrame));
```

1. IDR_SCRIBTYPE_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.
2. IDR_SCRIBTYPE_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.
3. 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);
```

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);
```

Alternatively you can register your application using one of the two methods described previously: manually merge the SCRIBBLE.REG registration file into the system registry, using REGEDIT, or programmatically merge the registration file from your application's setup program.

If the application was spawned by OLE as an in-place server or automation server, then `InitInstance` returns the call before performing additional initialization tasks that are appropriate only for stand-alone applications.

```
if (RunEmbedded() || RunAutomated())
{
    return TRUE;
}
```

Editing OLE-Related Resources

The next overall task is to:

- Add OLE standard resources
- Add menus
- Add toolbars
- Add accelerators for in-place active or fully opened servers

Add OLE Standard Resources

▶ **To add OLE standard resources**

1. Open SCRIBBLE.RC.
2. From the Resource menu, choose the Set Includes command.
3. Add the following to Compile-Time Directives:

```
    #include "afxprint.rc"  // printing/print preview resources
▶   #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. Choose OK to accept the changes you made in the Set Includes dialog box.
5. Choose 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_SCRIBTYPE is the menu for the document when it is opened in the usual way.
- IDR_SCRIBTYPE_SRVR_EMB is the menu for the document when the server is opened fully from the container application.
- IDR_SCRIBTYPE_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_SCRIBTYPE_SRVR_IP) looks like this:

  Edit | Pen | Help

  An example of a container's menu is seen in Contain (the OLE Container tutorial discussed in Chapters 17 through 19), as shown below:

  File | | Window

  OLE merges the two menus to create the following menu when the Scribble object is in-place activated in the Contain 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 `CScribApp::InitInstance`:

```
pDocTemplate->SetServerInfo(
    IDR_SCRIBTYPE_SRVR_EMB, IDR_SCRIBTYPE_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_SCRIBTYPE_SRVR_EMB and IDR_SCRIBTYPE_SRVR_IP, from the AppWizard-provided SCRATCH\SCRIBBLE.RC to MYSCRIB\SCRIBBLE.RC.
- Add the Pen pop-up menu to each of the menus.

▶ **To copy menu resources**

1. Open SCRIBBLE\MYSCRIB\SCRIBBLE.RC from the project window.
2. From the File menu, choose Open and select SCRIBBLE\SCRATCH\SCRIBBLE.RC.
3. Copy the menus IDR_SCRIBTYPE_SRVR_EMB and IDR_SCRIBTYPE_SRVR_IP from the scratch resource file to MYSCRIB\SCRIBBLE.RC.

▶ **To copy menu items**

1. Open the menu resources IDR_SCRIBTYPE and IDR_SCRIBTYPE_SRVR_EMB in the MYSCRIB resource file.
2. Copy the Pen pop-up menu from IDR_SCRIBTYPE into the IDR_SCRIBTYPE_SRVR_EMB menu so that it is between the Edit and View menus, as in IDR_SCRIBTYPE.
3. Copy the Clear All menu item from the Edit menu in IDR_SCRIBTYPE to the Edit menu in IDR_SCRIBTYPE_SRVR_EMB.
4. Repeat steps 1 to 3 for IDR_SCRIBTYPE_SRVR_IP, putting the Pen menu as follows:

   Edit | Pen | Help
5. Close the menu resource and save the resource file.

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 with four buttons. The **COleIPFrameWnd**-derived class is responsible for specifying the mapping of commands to toolbar buttons. AppWizard provides this in the **buttons** array in IPFRAME.CPP:

```
// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
          // same order as in the bitmap 'toolbar.bmp'
     ID_EDIT_CUT,
     ID_EDIT_COPY,
     ID_EDIT_PASTE,
         ID_SEPARATOR,
     ID_APP_ABOUT,
     ID_CONTEXT_HELP
};
```

▶ **To copy the toolbar bitmap resource**

You will use the bitmap editor, and follow the same kind of procedure that you did for menus.

1. Copy the toolbar bitmap resource, IDR_SCRIBTYPE_SRVR_IP, from the AppWizard-provided scratch resource file to MYSCRIB\SCRIBBLE.RC.
2. In MYSCRIB\SCRIBBLE.RC, copy the application-specific Pen Width button image from the IDR_MAINFRAME toolbar bitmap to the IDR_SCRIBTYPE_SRVR_IP toolbar bitmap, as follows:
   1. From the Image menu, choose Grid Settings, and select the Tile Grid option for both bitmap toolbar resources.
   2. Increase the width of the IDR_SCRIBTYPE_SRVR_IP bitmap from 80 to 96 pixels.
   3. Drag the Help and Context-Sensitive Help images to the last button position in the bitmap to make room for the Pen Width button image.
   4. Using Copy and Paste commands on the Edit menu, copy the Pen Width button image in IDR_MAINFRAME to IDR_SCRIBTYPE_SRVR_IP, and locate it between the Edit Paste button and the Help (question mark) button.

   This completes the editing of the toolbar bitmap, which should look like the one in Figure 16.1.

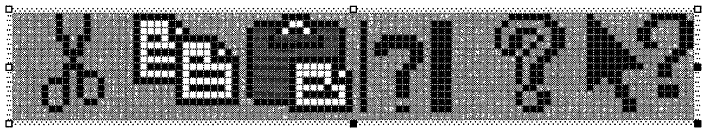


Figure 16.1 Toolbar bitmap

3. Using the text editor, insert the ID_PEN_THICK_OR_THIN command in the **buttons** array for the **COleIPFrameWnd** in IPFRAME.CPP:

```
// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
            // same order as in the bitmap 'itoolbar.bmp'
        ID_EDIT_CUT,
        ID_EDIT_COPY,
        ID_EDIT_PASTE,
▶         ID_SEPARATOR,
▶       ID_PEN_THICK_OR_THIN,
          ID_SEPARATOR,
        ID_APP_ABOUT,
      ..ID_CONTEXT_HELP,
};
```

The order of command IDs in the **buttons** array must correspond to the order of button images in the IDR_SCRIBTYPE_SRVR_IP toolbar bitmap resource. For more information on working with toolbar bitmap resources, see "Editing Toolbar Graphics" in Chapter 9, "Using the Graphic 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 pop-up 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_SCRIBTYPE_SRVR_EMB and IDR_SCRIBTYPE_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. Copy accelerator resources IDR_SCRIBTYPE_SRVR_EMB and IDR_SCRIBTYPE_SRVR_IP from SCRATCH\SCRIBBLE.RC to MYSCRIB\SCRIBBLE.RC.
2. Save your changes.

Adding Application-Specific Server Support

The rest of your task is to:

- 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:

- Implement notifying the OLE server that the embedded item has moved or changed size.
- Change the initial size of the document.
- Implement the document's support for putting a link format on the Clipboard.

▶ **To notify the OLE server when the embedded item moves or changes size**

1. Add these declarations to SCRIBDOC.H, as indicated by the marked lines:

```
// Implementation
    protected:
        void ReplacePen();
▶       COleServerItem* OnGetEmbeddedItem();
▶       void OnSetItemRects(LPCRECT lpPosRect,
▶           LPCRECT lpClipRect);
```

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, `CScribView::SetScrollInfo`, which you will call from the override of **OnSetItemRects**.

2. Add the marked lines to SCRIBDOC.CPP:

```
▶ COleServerItem* CScribDoc::OnGetEmbeddedItem()
▶ {
▶     // OnGetEmbeddedItem is called by the framework to get
▶     // the OleServerItem that is associated with the document.
▶     // It is only called when necessary.
▶
▶     CScribItem* pItem = new CScribItem(this);
▶     ASSERT_VALID(pItem);
▶     return pItem;
▶ }
▶
▶ void CScribDoc::OnSetItemRects(LPCRECT lpPosRect,
▶     LPCRECT lpClipRect)
▶ {
▶     // 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();
▶     CScribView* pView = (CScribView*)GetNextView(pos);
▶     pView->SetScrollInfo();
▶ }
```

3. Add the marked **#include** statement to SCRIBDOC.CPP, because the above implementation refers to `CScribView`:

```
  #include "pendlg.h"
▶ #include "scribvw.h"
▶ #include "scribitm.h"
```

The next step is to change Scribble's fixed document size, 8 by 9 inches, to 2 by 2 inches in `CScribDoc::InitDocument`. The current size, 8 by 9 inches, is too large for most containers.

▶ **To change the initial size of the document**

1. In SCRIBDOC.CPP, change:

```
void CScribDoc::InitDocument()
{
    ...
    //default document size
    ...
    m_sizeDoc = CSize(800, 900);
    ...
}
```

to:

```
▶ m_sizeDoc = CSize(200, 200);
```

2. Add the marked line to the `CScribDoc` constructor:

```
  CScribDoc::CScribDoc()
  {
      ...
▶     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 standalone 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. A good place to initialize `m_sizeDoc` is in the constructor.

▶ **To implement the document's support for putting a link format on the Clipboard**

With this support, containers can do the Paste Link command on the server's Edit menu.

1. Run ClassWizard.
2. Choose the Message Maps tab.
3. Select `CScribDoc` in the Class Name box.
4. Add the **COMMAND** handler for ID_EDIT_COPY, and accept the default name `OnEditCopy`.
5. Choose the Edit Code button, and add the marked lines to the `OnEditCopy` function.

```
  void CScribDoc::OnEditCopy()
  {
▶     CScribItem* 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 SCRIBITM.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. `CScribItem::OnDraw` needs to do essentially the same drawing that `CScribView::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.

▶ **To implement the OLE item's OnDraw function**

1. In SCRIBITM.CPP replace the AppWizard-provided stubbed version of `CScribItem::OnDraw` with:

```
BOOL CScribItem::OnDraw(CDC* pDC, CSize& rSize)
{
    CScribDoc* 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. Update the AppWizard-provided version of `CScribItem::OnGetExtent` with the code marked below:

```
  BOOL CScribItem::OnGetExtent(DVASPECT dwDrawAspect, CSize& rSize)
  {
▶     // This implementation of CScribItem::OnGetExtent only handles
▶     // the "content" aspect indicated by DVASPECT_CONTENT.

      if (dwDrawAspect != DVASPECT_CONTENT)
          return COleServerItem::OnGetExtent(dwDrawAspect, rSize);

      // CScribItem::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.
      CScribDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

▶     rSize = pDoc->GetDocSize();
▶     CClientDC dc(NULL);
▶
▶     // set a MM_LOENGLISH based on logical inches
▶     //  (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;
  }
```

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 above 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 visual editing server, it was lazy about its device context coordinates using **MM_LOENGLISH**: Scribble did not adjust for how many logical pixels per inch are 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. Choose the Message Maps tab.
3. Ensure that CScribView is chosen in the Class Name and Object IDs boxes.
4. Choose OnPrepareDC in the Member Functions box.
5. Choose Add Function.
6. Choose Edit Code.
7. Add the following code to the override of `OnPrepareDC` in SCRIBVW.H:

```
  void CScribView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
  {
▶     CScribDoc* 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);
    }
```

▶ **To adjust the scroll view's scroll bars to reflect the use of the logical MM_LOENGLISH mapping mode**

1. In SCRIBVW.H, declare the `SetScrollInfo` helper function:

```
    // Operations
    public:
▶       void SetScrollInfo();    // resync scroll sizes
```

2. In SCRIBVW.CPP, after `OnInitialUpdate`, implement `SetScrollInfo`:

```
▶   void CScribView::SetScrollInfo()
▶   {
▶       CClientDC dc(NULL);
▶       OnPrepareDC(&dc);
▶       CSize sizeDoc = GetDocument()->GetDocSize();
▶       dc.LPtoDP(&sizeDoc);
▶       SetScrollSizes(MM_TEXT, sizeDoc);
▶   }
```

3. In `OnInitialUpdate`, replace the call to **CScrollView::SetScrollSizes** with the call to the helper function, `SetScrollInfo`:

```
    void CScribView::OnInitialUpdate()
    {
▶       SetScrollInfo();
        CScrollView::OnInitialUpdate();
    }
```

4. Update the scroll bars appropriately when the window is sized.
   - Open ClassWizard.
   - Choose the Message Maps tab.
   - Select `CScribView` in the Class Names box.

- Select `CScribView` in the Object Types box and add the message handler for **WM_SIZE**.
- Choose the Edit Code button, and add the call to `SetScrollInfo` as indicated by the marked line:

```
  void CScribView::OnSize(UINT nType, int cx, cy)
  {
▶     SetScrollInfo();    //  ensure that scroll info is up-to-date
      CScrollView::OnSize(nType, cx, cy);
  }
```

5. Add the marked line to the `CScribView` constructor:

```
  CScribView::CScribView()
  {
▶     SetScrollSizes(MM_TEXT, CSize(0,0));
  }
```

It is necessary to initialize the scroll sizes in the **CScribView** constructor to default values, so that the extent is defined before the first call to `OnPrepareDC`.

When Scribble's view finished adding a new stroke in `CScribView::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 the additional call to **COleServerDoc::NotifyChanged.**

▶ **To notify OLE when the embedded item changes**

- In SCRIBVW.CPP, add the marked line to the **OnLButtonUp** function.

```
  void CScribView::OnLButtonUp(UINT, CPoint point)
  {
      ...

      ReleaseCapture();    // Release the mouse capture established at
                           // the beginning of the mouse drag.
▶     pDoc->NotifyChanged();
      return;
  }
```

Testing Scribble Server Functionality Using a Container Application

You have now completed all the work needed to convert Scribble to an OLE visual editing server application. Build the project in the usual way, and follow the steps in the preview of Scribble found at the beginning of the chapter. In particular, remember to run it once as a stand-alone. You have now converted Scribble to an OLE server that works as in our preview demonstration.

CHAPTER 17

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 Object Linking and Embedding (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, Contain. The Contain document can hold several OLE in-place items that the user can resize and move to any place in the document. However, the Contain 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 in Microsoft Foundation Class Library Samples.

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 the Chapters 6 through 15 before you begin this tutorial. The Scribble tutorial introduces important class library concepts and techniques, and it teaches you how to use the wizards and the resource editors.

Preview of the Contain Application

Before you work through the steps of implementing Contain, try out the completed application. This will help you appreciate OLE container functionality in general, and Contain's container functionality in particular, from a user's point of view.

Other example container applications, written using the OLE SDK rather than MFC, are provided on the Visual C++ distribution CD-ROM. You can run them from the Windows Program Manager group named "OLE Samples on CDROM."

The source code for these OLE samples is included on the distribution CD-ROM, and there is a Setup option that allows you to install the source code on your hard disk. Finally, the best examples are those world-class applications that support OLE today (not provided with Visual C++). These are the applications you will ultimately want to test your application against.

The first step is to install and register at least one OLE server application so that Contain will have access to at least one OLE server. A good example is the Step 7 version of Scribble. Another example is the HIERSVR sample. For more information on OLE containers and servers, see "OLE Overview: Containers and Servers" in *Programming with the Microsoft Foundation Class Library.*

▶ **To install and register the Scribble server application**

1. Build SCRIBBLE.EXE from SAMPLES\MFC\SCRIBBLE\STEP7, or run them from \SAMPLES\MFC\BIN on the CD.
2. Briefly run Scribble once so that it will register itself in your system as an OLE server.

▶ **To preview Contain**

1. Build CONTAIN.EXE from SAMPLES\MFC\CONTAIN\STEP2, and run it, or run it from \SAMPLES\MFC\BIN on the CD.
2. From the Edit menu, choose Insert New Object.
3. In the Insert Object dialog box, select SCRIB File Type from the Object Type box.

   A new Scribble drawing is created within the Contain document.
4. Choose OK.
   - Notice the tracker rectangle with resize handles and dashed border that appears in the upper-left corner of the Contain document. Contain negotiates with the server (Scribble) to determine where to place the initial rectangle and what size to make it.
   - Notice how Contain's menu has been merged with Scribble's (for example, notice the Pen menu) and how Contain's toolbar is replaced by one provided by Scribble (notice the Pen toolbar button).
5. Use the mouse to draw within the rectangle provided by the server running within the Contain container document. Try out Scribble's menu and toolbar commands in place; for example, try to change the pen width.
6. Click outside the Scribble object, somewhere else in the Contain 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 Contain document now has the focus.

7. 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.
8. Drag the Scribble object around and resize it.
9. From the Edit menu, choose Insert New Object to add additional OLE embedded objects.
10. Run Scribble, HIERSVR, or some OLE server stand-alone. Copy an object from the server to the Clipboard.
11. Paste the object from the Clipboard into the Contain 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: Contain

This tutorial consists of two steps. The sample program in SAMPLES\MFC\CONTAIN contains a subdirectory for each step: STEP1 and STEP2. Each step's subdirectory contains a Visual C++ project file, complete source files, and other files needed for the step. If you chose the MFC Samples option in Setup, these files will be installed on your hard disk with the same directory structure.

This tutorial shows you how to develop an MFC OLE container application that allows visual (in-place) editing. In Step 1 (Chapter 18), 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 19), 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 18

Implementing Basic OLE Container Features

Step 1 of the OLE container tutorial shows you how to create an OLE container application. 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

▶ **To create a skeleton OLE container-enabled application**

1. From the File menu, choose New.

   The New dialog box appears.
2. Select Project.

   The New Project dialog box appears.
3. In the Project Name box, type **contain.**

   The application's project file will be given this name: in this case, CONTAIN.MAK.

Note If you have Visual C++ for Macintosh, please clear the Macintosh check box in the Platforms box. OLE support, as required by this tutorial, will not be generated if this check box is selected.

4. In the New Subdirectory box, delete "contain" and type **mycontr**.
5. Specify the path to the project's subdirectory.

   Use the list box provided to navigate through the directories on the selected drive. As you navigate through the directory structure, the path listed in the dialog box changes to show where the named subdirectory (MYCONTR) should be placed. When the path suits you, stop navigating.

   For Scribble, navigate to SAMPLES\MFC\CONTAIN. Assuming your Visual C++ installation is in directory MSVC20 on drive C, the path should look like this in the dialog box:

   ```
   c:\msvc20\samples\mfc\contain\mycontr\contain.mak
   ```

6. Choose the Create button.

   The MFC AppWizard Step 1 dialog box appears.
7. Choose the Next button 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.*
8. In the AppWizard Step 3 dialog box, select Container.

   Do not select Automation Support because you will not be adding automation support to Contain.
9. Choose Next.
10. In AppWizard Step 4 dialog box, choose the Advanced button.

    The Advanced Options dialog box appears.
11. Choose the Document Template Strings tab.
    - In the File Extension box, type **cnt** without a period.

      The correct information appears in the File Filter box.
    - In the Doc Type Name box, change "Contai" to "Contr."
12. Choose Close and Next.
13. In AppWizard Step 5, choose the Next button 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.*

14. In the AppWizard Step 6 dialog box, check and modify class names and filenames. Some of the class names must be changed from the defaults that AppWizard suggests. To edit the information for a class, select the class name in the box at the top of the dialog box.
    - Select the class `CContainDoc`. Change its header filename to **contrdoc.h** and the implementation filename to **contrdoc.cpp**.
    - Select the class `CContainView`. Change its header filename to **contrview.h** and the implementation filename to **contrview.cpp**.
    - Select the class `CContainCntrItem`. Change its name to **CCntrItem**.
15. Choose Finish.

    The New Project Information dialog box appears.
16. Choose OK.

    AppWizard creates the starter files for Contain.
17. From the Project menu, choose Build CONTAIN.exe.

For more information on OLE options in AppWizard, see the article "AppWizard: OLE Support" in *Programming with the Microsoft Foundation Class Library.*

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. Notice that it already has many of the features you saw in the preview of the Contain application, but several other features are implemented in this tutorial.

▶ **To create a new Scribble drawing within Contain**

1. From the Edit menu, choose Insert New Object.
2. In the Insert Object dialog box, select the "Scrib Document" from the Object Type box.
3. Choose 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 the Microsoft Foundation Class Library.*

   Notice how Contain's menu is merged with Scribble's and how Contain'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 Contain does not yet support hit testing and selection.
4. Press the ESC key 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 Contain 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.

   - From the Edit menu, choose "Scrib Object" and then choose Edit.

     The Edit verb activates the object in place. (The Open verb fully opens the Scribble server.) The Scribble object is reactivated in place, which is indicated by the tracker. For more information on OLE verbs, see the article "Activation: Verbs" in *Programming with the Microsoft Foundation Class Library*.
   - Note that the Scribble server provides the tracker; Contain provides the tracker only when the object is not activated in place.
6. Exit Contain.

In summary, the AppWizard-provided container application already has a lot of OLE container functionality, but it is still missing some basic functionality that is 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 the Microsoft Foundation Class Library*.

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 what code you need to manually add to your application.

CContainApp

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 the Microsoft Foundation Class Library.*

  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.

CContainView

The member `CCntrItem* 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 simplifying 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.

```
void CContainView::OnDraw(CDC* pDC)
{
...

    if (m_pSelection == NULL)
    {
        POSITION pos = pDoc->GetStartPosition();
        m_pSelection = (CCntrItem*)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 Contain application, which has a simple single-selection user interface. For an example of multiple selection, see the Drawcli sample.

```
BOOL CContainView::IsSelected(const CObject* pDocItem) const
{
    // The implementation below is adequate if your selection consists
    //  of only CCntrItem 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 the Microsoft Foundation Class Library.*

```
void CContainView::OnInsertObject()
{
    // Invoke the standard Insert Object dialog box to obtain
    //  information for new CCntrItem object.
    COleInsertDialog dlg;
    if (dlg.DoModal() != IDOK)
        return;

    BeginWaitCursor();

    CCntrItem* pItem = NULL;
    TRY
    {
        // Create new item connected to this document.
        CContainDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pItem = new CCntrItem(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 **CContainView::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 Contain and by most container applications.

```
void CContainView::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 `CContainView::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 CContainView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    COleClientItem* pActiveItem
        = GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->SetItemRects();
}
```

CCntrItem

Class `CCntrItem` 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 `CContainView::OnInsertObject` provided by AppWizard does create the `CCntrItem` object, as explained earlier. The application explicitly deletes a **COleClientItem** object only in certain cases, such as when the user presses the DEL key when this item is selected or when the entire containing document is destroyed. For more information, see the articles "Containers: Client Items" in *Programming with the Microsoft Foundation Class Library.*

The AppWizard-provided implementation of `CCntrItem::OnChange` simply calls **OnChange** in the base class, **COleClientItem**, and then, just to be safe, invalidates all views of the document.

```
void CCntrItem::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 CCntrItem::OnGetItemPosition(CRect& rPosition)
{
    ASSERT_VALID(this);

    // During in-place activation, CCntrItem::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
    //  CCntrItem::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 CCntrItem::OnChangeItemPosition(const CRect& rectPos)
{
    ASSERT_VALID(this);

    // During in-place activation CCntrItem::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 `CCntrItem` 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 CNTRITEM.H.

   The AppWizard-provided implementation of `CCntrItem` 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:

```
//Attributes
public:
▶   CRect m_rect;  // position within the document
```

2. Initialize `m_rect` in the `CCntrItem` constructor in CNTRITEM.CPP.

```
CCntrItem::CCntrItem(CContainDoc* pContainer)
    : COleClientItem(pContainer)
{
▶   m_rect.SetRect(10, 10, 50, 50);
}
```

3. Replace the implementation of `OnGetItemPosition` with the marked lines.

```
  void CCntrItem::OnGetItemPosition(CRect& rPosition)
  {
▶     ASSERT_VALID(this);
▶
▶     // 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 `CCntrItem` item is now being tracked by **CRectTracker**, the framework's request is satisfied by returning the **CRect** member, `m_rect`.

   In Contain Step 2, this implementation is replaced with one that allows the server to negotiate the size of the object.

4. Complete the implementation of `OnChangeItemPosition`.

   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 `CCntrItem` 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.

```
BOOL CCntrItem::OnChangeItemPosition(const CRect& rectPos)
{
    ASSERT_VALID(this);

    // During in-place activation CCntrItem::OnChangeItemPosition
    //  is called by the server to change the position on 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;
▶       GetDocument()->UpdateAllViews(NULL);
▶       m_rect = rectPos;
▶
▶       // mark document as dirty
▶       GetDocument()->SetModifiedFlag();

        return TRUE;
    }
```

In Contain Step 2, the simple **UpdateAllViews** call is replaced with smart invalidation.

5. Serialize the **CRect** `m_rect` member variable in `CCntrItem::Serialize` as indicated by the marked lines:

```
    if (ar.IsStoring())
    {
▶       ar << m_rect;
    }
    else
    {
▶       ar >> m_rect;
    }
```

Implementing Hit Testing and Selection

The AppWizard-provided skeleton application initially supports only one embedded object. Support for multiple objects can now be added by implementing hit testing and selection.

▶ **To implement hit testing**

1. Declare the helper function `HitTestItems` in CONTRVW.H.

```
    // Operations
    public:
▶       CCntrItem* HitTestItems(CPoint point);
```

2. Implement the function `HitTestItems` in CONTRVW.CPP after **CContainView::OnSize**.

   Hit testing determines which of the multiple objects lies at a given point.

```
▶   CCntrItem* CContainView::HitTestItems(CPoint point)
▶   {
▶       CContainDoc* pDoc = GetDocument();
▶       CCntrItem* pItemHit = NULL;
▶       POSITION pos = pDoc->GetStartPosition();
▶       while (pos != NULL)
▶       {
▶           CCntrItem* pItem = (CCntrItem*)pDoc->GetNextItem(pos);
▶           if (pItem->m_rect.PtInRect(point))
▶               pItemHit = pItem;
▶       }
▶       return pItemHit; // return top item at point
▶   }
```

▶ **To implement selection**

1. Declare the helper function `SetSelection` in CONTRVW.H.

```
// Operations
public:
    ...
▶       void SetSelection(CCntrItem* pItem);
```

2. Implement `SetSelection` in CONTRVW.CPP after the implementation of the `HitTestItems` member function.

```
▶ void CContainView::SetSelection(CCntrItem* pItem)
▶ {
▶     // 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;
▶ }
```

The above implementation is "lazy" in that it invalidates the entire client area of the view whenever the selection changes. In Contain Step 2, this implementation is replaced with smarter invalidation.

Implementing Activation by Using a Mouse Click

Contain 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 the Microsoft Foundation Class Library*.

Implementing the **OnLButtonDown** handler so that a single click selects the embedded object follows this scheme:

- Call `HitTestItems` to find the `CCntrItem` at the point where the mouse was clicked.
- Set the selection to be this `CCntrItem`. Note, if no `CCntrItem` 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 and allows the user to drag the tracker rectangle around on the screen and allows the user 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 the Microsoft Foundation Class Library.*
- If the user has resized the object (indicated by a value of **TRUE** being returned from **CRectTracker::Track**), update the `m_rect` of the `CCntrItem` object.

▶ **To implement the OnLButtonDown mouse handler**

1. Use ClassWizard to add **WM_LBUTTONDOWN** handler for `CContainView`.
2. Implement `CContainView::OnLButtonDown` in CONTRVW.CPP.

   For now, the entire client area of the view is invalidated. Smarter invalidation is implemented in Step 2.

```
  void CContainView::OnLButtonDown(UINT nFlags, CPoint point)
  {
▶     CCntrItem* 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();
▶           }
▶       }
▶
        CView::OnLButtonDown(nFlags, point);
    }
```

3. Declare the helper function `CContainView::SetupTracker` in CONTRVW.H:

```
    // Operations
    public:
    ...
        void SetSelection(CCntrItem* pItem);
▶       void SetupTracker(CCntrItem* pItem, CRectTracker* pTracker);
```

4. Implement the helper function, `CContainView::SetupTracker` after the implementation of the `SetSelection` member function. `SetupTracker` sets up the styles of the tracker rectangle according to the state of the `CCntrItem` object, such as whether it has been selected.

```
▶   void CContainView::SetupTracker(CCntrItem* pItem,
▶       CRectTracker* pTracker)
▶   {
▶       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 the CTRL key 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 ClassWizard to add the **WM_LBUTTONDBLCLK** handler to **CContainView.**
2. Implement `CContainView::OnLButtonDblClk` in CONTRVW.CPP by adding the indicated lines:

```
  void CContainView::OnLButtonDblClk(UINT nFlags, CPoint point)
  {
▶     OnLButtonDown(nFlags, point);
▶
▶     if (m_pSelection != NULL)
▶     {
▶         m_pSelection->DoVerb(GetKeyState(VK_CONTROL) < 0 ?
▶             OLEIVERB_OPEN : OLEIVERB_PRIMARY, this);
▶     }
▶
      CView::OnLButtonDblClk(nFlags, point);
  }
```

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 ClassWizard to add a **WM_SETCURSOR** handler for `CContainView`.
2. Implement the `CContainView::OnSetCursor` handler.

```
  BOOL CContainView::OnSetCursor(CWnd* pWnd, UINT nHitTest,
       UINT message)
  {
▶     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;
▶     }
▶
      return CView::OnSetCursor(pWnd, nHitTest, message);
  }
```

Drawing the Embedded Objects

The AppWizard-provided implementation of CContainView::OnDraw simply draws the one embedded object pointed to by m_pSelection. Now that Contain supports multiple embedded objects, OnDraw must be reimplemented accordingly.

▶ **To support drawing of multiple embedded objects**

- Replace the implementation of CContainView::OnDraw provided by AppWizard with the marked lines:

```
  void CContainView::OnDraw(CDC* pDC)
  {
      CContainDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

▶     // draw the OLE items from the list
▶     POSITION pos = pDoc->GetStartPosition();
▶     while (pos != NULL)
▶     {
▶         // draw the item
▶         CCntrItem* pItem = (CCntrItem*)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 &Link

      &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.
2. Save the resource file.
3. Open ClassWizard.
4. Choose the Message Maps tab.
5. In the Class Name box, select `CContainView` if it is not already selected.
6. Create a pair of **ON_COMMAND** and **ON_UPDATE_COMMAND_UI** handlers for ID_EDIT_CLEAR. Accept the default handler function names `OnEditClear` and `OnUpdateEditClear`.

7. Choose the Edit Code button and implement both `OnEditClear` and `OnUpdateEditClear` as follows:

```
   void CContainView::OnEditClear()
   {
▶      if (m_pSelection != NULL)
▶      {
▶          m_pSelection->Delete();
▶          m_pSelection = NULL;
▶          GetDocument()->UpdateAllViews(NULL);
▶      }
   }

   void CContainView::OnUpdateEditClear(CCmdUI* pCmdUI)
   {
▶      pCmdUI->Enable(m_pSelection != NULL);

   }
```

To delete an embedded object in a container, simply call the object's **COleClientItem::Delete** function.

Building and Running Contain Step 1

Build Contain 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 Contain document. When the Insert Object dialog box appears, select Scrib Document in the Object Type box.

  Notice that the Scribble object initially has a size of (10, 10, 50, 50), as determined by the `CCntrItem` constructor. Contain 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.
- Close HIERSVR.
- From the Edit menu, choose Insert New Object.
- Select the Create from File option.
- Type the name of the HIERSVR file just created.
- Select the Link check box.
- Press 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 19, 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 19

Refining OLE Container Functionality

As implemented in Step 1, Contain 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 Contain 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 Contain. For that reason, you should save the Step 1 version of CONTAIN.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 Contain'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 ClassWizard.
2. Choose the Message Maps tab.
3. Select `CContainView` in the Class Name box.
4. Add both the **COMMAND** and **UPDATE_COMMAND_UI** handlers for ID_EDIT_COPY, and accept the default function names, `OnEditCopy` and `OnUpdateEditCopy`, respectively.

5. Add the marked lines to implement the Copy command on the Edit menu:

```
  void CContainView::OnEditCopy()
  {
▶     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.

6. Add the marked code to update the active selection.

```
  void CContainView::OnUpdateEditCopy(CCmdUI* pCmdUI)
  {
▶     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. Open ClassWizard.
2. Choose the Message Maps tab.
3. In the Class Name box, select `CContainView`.
4. Add the **COMMAND** handler for just ID_EDIT_PASTE, and accept the default function name, `OnEditPaste`.
5. Add the marked lines to implement the Paste command on the Edit menu:

```
  void CContainView::OnEditPaste()
  {
▶     CCntrItem* pItem = NULL;
▶
▶     TRY
▶     {
▶         // Create new item connected to this document.
▶         CContainDoc* pDoc = GetDocument();
▶         ASSERT_VALID(pDoc);
▶         pItem = new CCntrItem(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 the code for constructing a new `CCntrItem`.

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
- Invalidate the tracked object

Define the Update Hint

The first task is to define the update hint.

▶ **To define the update hint**

- Define the following two **#define** values in CONTRDOC.H:

```
▶ #define HINT_UPDATE_WINDOW        0
▶ #define HINT_UPDATE_ITEM          1

  class CContainDoc : public COleDocument
  // ...
```

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 Contain.

▶ **To receive the hint and invalidate the view**

1. Open ClassWizard.
2. Choose the Message Maps tab.
3. Select the CContainView class.
4. Select "CContainView" in the Object IDs box.
5. Select "OnUpdate" in the Member Functions box.

6. Choose the Edit Code button.

   The Edit Code button transfers you to the text editor, opens CONTRVW.CPP, and displays the definition for `OnUpdate`.

7. Add the marked code to the `OnUpdate` function definition:

```
  void CContainView::OnUpdate(CView* pSender, LPARAM lHint,
                              CObject* pHint)
  {
▶     switch (lHint)
▶     {
▶     case HINT_UPDATE_WINDOW: // invalidate entire window
▶         Invalidate();
▶         break;
▶     case HINT_UPDATE_ITEM:      // invalidate single item
▶         {
▶             CRectTracker tracker;
▶             SetupTracker((CCntrItem*)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 `CContainView::OnUpdate` takes this into account.

The two **#define** HINT_ values are used for the **LPARAM** *lHint* value passed to `CContainView::OnUpdate`. 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.

Centralize the Sending of Update Hints

There are several occasions when Contain needs to send the **HINT_UPDATE_ITEM** hint for a `CCntrItem` 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 `CCntrItem`. Thus, it makes sense to have the `CCntrItem` object itself send the update hint by using the document's `UpdateAllViews` function. `CCntrItem::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 CCntrItem object**

1. Declare a new public member function `CCntrItem::InvalidateItem` in CNTRITEM.H:

```
  // Attributes
  // ...
▶ // Operations
▶ public:
▶     void InvalidateItem();
```

2. To implement `CCntrItem::InvalidateItem`, add the marked lines to CNTRITEM.CPP, after **CCntrItem::~CCntrItem**:

```
▶ void CCntrItem::InvalidateItem()
▶ {
▶     GetDocument()->UpdateAllViews(NULL, HINT_UPDATE_ITEM, this);
▶ }
```

   Note that the framework keeps track of which document object owns the `CCntrItem` object and therefore implements `CCntrItem::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**

- In the implementation of `CContainView::SetSelection` in CONTRVW.CPP, replace the following code:

```
Invalidate();
m_pSelection = pItem;
```

  with the code marked below:

```
void CContainView::SetSelection(CCntrItem* pItem)
{
    // close in-place active item
    if (pItem == NULL || m_pSelection != pItem)
    {
        COleClientItem* pActiveItem
            = GetDocument()->GetInPlaceActiveItem(this);
        if (pActiveItem != NULL && pActiveItem != pItem)
            pActiveItem->Close();
    }
▶       // 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**

- In the implementation of `CContainView::OnLButtonDown` in CONTRVW.CPP, replace:

```
Invalidate();
pItemHit->m_rect = tracker.m_rect;
```

with:

```
▶ pItemHit->InvalidateItem();
▶ pItemHit->m_rect = tracker.m_rect;
▶ pItemHit->InvalidateItem();
```

Invalidate Object Moved by 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 CCntrItem object changes**

- In the implementation of `CCntrItem::OnChangeItemPosition` in CNTRITEM.CPP, replace:

```
GetDocument()->UpdateAllViews(NULL);
m_rect = rectPos;
```

with the marked lines:

```
▶ InvalidateItem();
▶ m_rect = rectPos;
▶ InvalidateItem();
```

Coordinating with Server to Determine Size of Object

The following exercise demonstrates why Contain 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 the Microsoft Foundation Class Library.*

Demonstration

1. If you have not already built the HIERSVR sample, do that now. Run HIERSVR once to register this OLE server application with OLE and then close it.
2. Run Step 1 of Contain.
3. From the Edit menu, choose Insert New Object. 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 `CCntrItem` constructor. Contain does not give HIERSVR the opportunity to set the initial size of the object.
4. From HIERSVR's Edit menu, choose Add Node to add a second node.

   Notice that Contain correctly increases the height of the object to accommodate the new node. You can add more nodes, and Contain continues to increase the height of the object. In Step 1 the implementation of `CCntrItem::OnChangeItemPosition` changes the height of the in-place activated object at the request of HIERSVR.
5. Deactivate the HIERSVR object, then click once to select it.
6. From the Edit menu, choose Hierarchy List Object, then choose Open from the submenu.

   This fully opens the HIERSVR server application. Arrange HIERSVR and Contain on the screen so you can see both applications at the same time.
7. In HIERSVR, from the Edit menu choose Add Node to add another node.

   Notice that the size of the object in Contain 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.

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 `CCntrItem::OnChange` instead of `OnChangeItemPosition`. At this time, Contain needs to find out the new natural size of the object from HIERSVR. It does this by calling **COleClientItem::GetExtent**.

COleClientItem::GetExtent 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 `CCntrItem::OnChange` function is not the only place where Contain needs to call **COleClientItem::GetExtent** to get the natural extent of the object and then set the `m_rect` of the `CCntrItem`. Therefore, you will implement the helper function `UpdateFromServerExtent` as described below.

Get the Extent of the CCntrItem Object from the Server

To get the extent of the `CCntrItem` object from the server, and update the `m_rect` of the container item, implement the helper function `CCntrItem::UpdateFromServerExtent`.

▶ To get the extent of a CCntrItem object

1. Add the declaration of `UpdateFromServerExtent` to CNTRITEM.H as indicated by the marked line:

```
  // Operations
  public:
      void InvalidateItem();
▶     void UpdateFromServerExtent();
```

2. Implement the helper function by adding the marked lines to CNTRITEM.CPP, after **CCntrItem::InvalidateItem**.

```
▶ void CCntrItem::UpdateFromServerExtent()
▶ {
▶     CSize size;
▶     if (GetExtent(&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 CCntrItem 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 `CCntrItem` rectangle needs to be updated.

▶ **To update the CCntrItem rectangle when the item's natural extent changes**

- Replace the AppWizard-provided implementation of `OnChange` in CNTRITEM.CPP with the marked lines:

```
void CCntrItem::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.
▶       switch (nCode)
▶       {
▶       case OLE_CHANGED:
▶           InvalidateItem();
▶           UpdateFromServerExtent();
▶           break;
▶       case OLE_CHANGED_STATE:
▶       case OLE_CHANGED_ASPECT:
▶           InvalidateItem();
▶           break;
▶       }
    }
```

Notice that the `CCntrItem` 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 Contain, 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**

- In CONTRVW.CPP, insert the marked lines in `CContainView::OnInsertObject`:

```
  // Create new item connected to this document.
  CContainDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
  pItem = new CCntrItem(pDoc);
  ASSERT_VALID(pItem);

  // Initialize the item from the dialog data.
  if (!dlg.CreateItem(pItem))
      AfxThrowMemoryException();  // any exception will do
  ASSERT_VALID(pItem);

▶ pItem->UpdateLink();
▶ pItem->UpdateFromServerExtent();

      // If item created from class list (not from file)
```

COleClientItem::UpdateLink is called so that if the server is fully open, Contain 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 CONTRVW.CPP, replace the code in **CContainView::OnInsertObject**:

```
...
    // 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 Contain. It now performs exactly as you previewed it in Chapter 17.

CHAPTER 20

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 Disptest (Dispatch Test) tool included in Visual C++ as part of the OLE Toolkit.

If you want to learn about implementing an OLE automation client application, look at the CALCDRIV sample application and its description in the Microsoft Foundation Class Library Samples, under *Microsoft Foundation Class Library* in the Contents browser.

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 6 through 15 before you begin this tutorial. The Scribble tutorial introduces important class library concepts and techniques, and it teaches you how to use the wizards and the resource editors.

The Tutorial Example: Autoclik

In this tutorial you will create a simple OLE automation server application, Autoclik. 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 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 text 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.

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 properties and methods. 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 Disptest tool included as part of the OLE Toolkit with Visual C++. The Disptest tool is a simplified version of Microsoft Visual Basic. The following preview of Autoclik illustrates how you can drive an automation server using Disptest or Visual Basic. Later in this chapter, you will briefly explore how to write the Visual Basic code you can run from Disptest or Visual Basic.

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. An OLE automation server must be registered before it can be driven by any automation client, just as with OLE object servers.

▶ **To install and register the Autoclik automation server**

1. Build AUTOCLIK.EXE from SAMPLES\MFC\AUTOCLIK\STEP3, or run Autoclik from the \SAMPLES\MFC\BIN directory on the Visual C++ distribution CD-ROM.
2. Run Autoclik once so it will register itself as an OLE server.

▶ **To preview Autoclik**

1. In the "OLE Toolkit" Program Manager group installed by Visual C++ Setup, double-click the Automation Test icon, which runs DISPTEST.EXE. An alternative is to run Microsoft Visual Basic version 3.0.
2. From the File menu of Disptest or Visual Basic, choose Open Project and specify the following path:

   SAMPLES\MFC\AUTOCLIK\AUTODRIV.MAK.

   AUTODRIV.MAK is the Visual Basic project file for Autodriv, a small application specifically written to test drive Autoclik.
3. From the Run menu, choose Start. This brings up the Autoclik Test Driver dialog box shown in Figure 20.1.

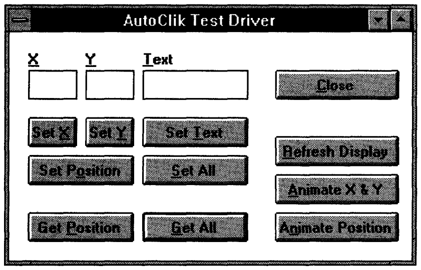


Figure 20.1 Autoclik Test Driver Dialog Box

4. Bring Autoclik into view next to Autodriv's window. Autodriv launches Autoclik on startup (Figure 20.2), but Autoclik might initially be hidden behind some of Disptest's or Visual Basic's windows. To keep both Autoclik and Autodriv visible, minimize any windows you won't need for this tutorial.

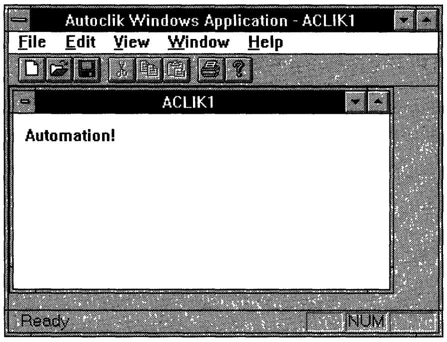

Autoclik Windows Application - ACLIK1
File Edit View Window Help
ACLIK1
Automation!
Ready
NUM

Figure 20.2 The Autoclik Application

▶ **To explore the automation server features of Autoclik**

The `X`, `Y`, and `Text` fields in Autodriv's window are initially blank.

1. Choose the Get All button.

   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 way 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 call back/notification interfaces be established so that the automation client can implement them.

3. From Autoclik's Edit menu, choose Change Text to change the text to "hello."
4. Choose the Get All button in Autodriv.

   It now shows new `X`, `Y`, and `Text` values.

5. Change the `X`, `Y`, and `Text` fields in Autodriv, and then choose the Set All button.

   Autoclik accepts the changes.

6. 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.

Command	Description
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.
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. The AUTOCLIK sample program directory in your SAMPLES\MFC directory contains a subdirectory for each step: STEP1, STEP2, and STEP3. Each step's subdirectory contains a Visual C++ project file, complete source files, and other files needed for the step. If you chose the MFC Samples option in Setup, these files are installed on your hard disk with the same directory structure.

In Step 1 (Chapter 21), 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 22), you implement the document dispatch interface. You will learn how to use ClassWizard to:

- Expose the `CClikDoc` member variable `m_pt` by using the Get and Set methods in Autoclik's document dispatch interface.
- Expose the `CClikDoc` 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 23), 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 21

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

▶ **To create a skeleton OLE automation server application**

1. From the File menu, choose New.

   The New dialog box appears.
2. Select Project.

   The New Project dialog box appears.
3. In the Project Name box, type **autoclik.**

   The application's project file will be given this name: in this case, AUTOCLIK.MAK.

   Note If you have Visual C++ for Macintosh, please clear the Macintosh check box in the Platforms box. OLE support, as required by this tutorial, will not be generated if this check box is selected.

   ---
4. In the New Subdirectory box, delete "autoclik" and type **myauto.**

5. Specify the path to the project's subdirectory.

   Use the list box provided to navigate through the directories on the selected drive. As you navigate through the directory structure, the path listed in the dialog box changes to show where the named subdirectory (MYAUTO) should be placed. When the path suits you, stop navigating. Assuming your Visual C++ installation is in directory MSVC20 on drive C, the path should look like this in the dialog box:

   ```
   c:\msvc20\samples\mfc\autoclik\myauto\autoclik.mak
   ```

6. Choose the Create button.

   The MFC AppWizard Step 1 dialog box appears.

7. Choose the Next button 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.*

8. In the AppWizard Step 3 dialog box, select the Yes button for "Would you like support for automation?"

   Do not select any server or container options because you will not be adding that type of functionality to Autoclik.

9. Choose Next.
10. In AppWizard Step 4 dialog box, choose the Advanced button.

    The Advanced Options dialog box appears.

11. Choose the Document Template Strings tab.
    - In the Doc Type Name box, change "Autocl" to "ACLIK."
    - In the File Extension box, change the proposed "aut" extension to **ack** without a period.

      The correct information appears in the File Filter box.

12. Choose Close.
13. In AppWizard Step 5, choose the Next button 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.*

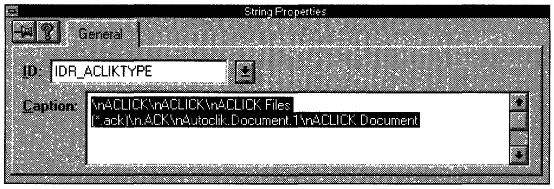
14. In the AppWizard Step 6 dialog box, check and modify class names and filenames. Some of the class names can be simplified from the defaults that AppWizard suggests. To edit the information for a class, select the class name in the box at the top of the dialog box.
    - Select the class `CAutoclikApp`, and change its name to **CClikApp**.
    - Select the class `CAutoclikDoc`, and change its name to **CClikDoc**.
    - Select the class `CAutoclikView`, and change its name to **CClikView**.
15. Choose Finish.
16. Choose the Create button in the New Project Information dialog box after examining the specifications in the box.

    AppWizard provides the starter files for your new application. There is no need for you to build it yet.

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.

A dispatch interface name is a literal string that automation clients use to access the automation server. If you open the string editor, you can look at, or change, these strings. The string ID is the IDR_<*yourapp*>TYPE created by AppWizard, which is registered in the application's `InitInstance`. For Autoclik, the string ID is IDR_ACLIKTYPE, and it can be found in Segment 0 of the string table. The strings for IDR_ACLIKTYPE are shown in Figure 21.1.



ID: IDR_ACLIKTYPE

Caption:
```
\nACLICK\nACLICK\nACLICK Files
(*.ack)\n.ACK\nAutoclik.Document.1\nACLICK Document
```

Figure 21.1 IDR_ACLIKTYPE in the String Editor

This string resource consists of several strings separated by newline characters (`\n`). The string `"Autoclik.Document"` is the document's dispatch interface name provided by AppWizard. The Visual Basic application that test drives Autoclik refers to this dispatch interface name in the following code:

```
Sub Form_Load()
    Set clik = CreateObject("Autoclik.Document")
    clik.ShowWindow
End Sub
```

The naming convention you used for Autoclik's document dispatch interface is *<appname>*.Document as in `"Autoclik.Document"`. As an application evolves, it might need to distinguish dispatch interfaces with a version number, as in "Autoclick.Document.1". By default, AppWizard creates a version independent identifier.

You may prefer a document dispatch interface name different from the one AppWizard provides. To change it, use the string editor to replace the text in the document template string resource. At the same time, you might want to change the other document template strings. For more information about them, see the documentation for **CDocTemplate::GetDocString** in the *Class Library Reference*.

Note An automation server may have more than one dispatch interface. Autoclik will have two dispatch interfaces. The initial AppWizard-created application has only one dispatch interface, 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 the 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 **COleTemplate::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 AUTOCDOC.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(CClikDoc)
    // 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(CClikDoc, CDocument)
    //{{AFX_DISPATCH_MAP(CClikDoc)
    // 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:

```
CClikDoc::CClikDoc()
{
    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 AutoServer to start, the server will exit when the object goes out of scope in the client.

```
CClikDoc::~CClikDoc()
{
    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 dispatch interfaces provided by the automation server, and for each dispatch interface, a list of properties and methods exposed by the automation server.

Whenever you define new dispatch interfaces, and define new methods and properties for the automation server, ClassWizard adds information to the .ODL file. When you build the application, the IDE spawns MKTYPLIB.EXE to create an updated .TLB file.

A good example of how the type library file is used is the Read Type Library option of ClassWizard itself. 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 dispatch interface 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. If you want to simply read along for the rest of Step 1 and start working on Step 2, copy the completed \MSVC20\SAMPLES\MFC\AUTOCLIK\STEP1 sources to your AUTOCLIK\MYAUTO subdirectory.

▶ **To declare the member variables of Autoclik's document class**

1. Open AUTOCDOC.H.
2. Declare the following:

```
   // Attributes
   public:
▶      CPoint m_pt;
▶      CString m_str;
```

▶ **To initialize the member variables of the document class**

1. Open AUTOCDOC.CPP.
2. Add the marked lines to the constructor:

```
   CClikDoc::CClikDoc()
   {
       EnableAutomation();

▶      m_pt = CPoint(10,10);
▶      m_str = _T("Automation!");

       AfxOleLockApp();
   }
```

3. Serialize the member variables in the document class.

```
   void CClikDoc::Serialize(CArchive& ar)
   {
       if (ar.IsStoring())
       {
▶          ar << m_pt << m_str;
       }
       else
       {
▶          ar >> m_pt >> m_str;
       }
   }
```

▶ **To implement Autoclik's drawing code**

1. Open AUTOCVW.CPP.
2. Implement `OnDraw` by adding the marked line:

```
  void CClikView::OnDraw(CDC* pDC)
  {
      CClikDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

▶     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. Declare `Refresh` as a public member function in AUTOCDOC.H:

```
  \\ Operations
  public:
▶     void Refresh();
```

2. Implement it in AUTOCDOC.CPP, after **CClikDoc::OnNewDocument**, as:

```
▶ void CClikDoc::Refresh()
▶ {
▶     UpdateAllViews(NULL);
▶     SetModifiedFlag();
▶ }
```

▶ **To implement the mouse click handler**

1. Open ClassWizard
2. Choose the Message Maps tab.
3. In the Class Name box, select `CClikView`.
4. In the Object ID box, select `CClikView`.
5. Double-click **WM_LBUTTONDOWN** in the Message box.
6. Choose the Edit Code button.

7. Implement the mouse click handler by adding the marked lines:

```
   void CClikView::OnLButtonDown(UINT nFlags, CPoint point)
   {
▶      CClikDoc* pDoc = GetDocument();
▶      pDoc->m_pt = point;
▶      pDoc->Refresh();

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

▶ **To implement the Change Text dialog box**

1. In the project window, double-click AUTOCLIK.RC.
2. From the Resource menu, choose New.

   The New Resource dialog box appears.
3. Select Dialog and choose OK..
4. Type the following information in the dialog's property page:
   - In the Dialog identification box, type **IDD_CHANGE_TEXT**.
   - In the Caption box, type **Change Text**.
5. Add a static text control labeled: "Enter Text:" and add an Edit control for the text.
6. Open ClassWizard. In the Add Class dialog box, type the following information:
   - In the Class Name box: **CChangeText**
   - In the Header File box: **dialogs.h**
   - In the Implementation box: **dialogs.cpp**
7. Choose the Create Class button.
8. Choose the Member Variables tab in Class Wizard.
9. Double-click IDC_EDIT1, and type **m_str** in the Member Variable Name box to add the member variable for the edit control.
10. Choose OK twice.

▶ **To add the Change Text command to Autoclik's Edit menu**

1. In the resource browser window, double-click Menu.
2. Double click IDR_ACLIKTYPE.

   The menu editor opens
3. Click Autoclik's Edit menu.
4. Add a separator below the Paste menu item.

5. Add the following menu item text below the separator:

   Change &Text...

6. The menu editor automatically names the command ID_EDIT_CHANGETEXT.
7. Type a prompt string such as

   ```
   Change text displayed in the view.
   ```

▶ **To implement the handler for the Change Text command**

1. Open ClassWizard.
2. Choose the Message Maps tab.
3. In the Class Name box, select `CClikDoc`.
4. In the Object ID box, select ID_EDIT_CHANGETEXT.
5. Double-click **COMMAND**, and accept the default name `OnEditChangetext`.
6. Choose the Edit Code button.
7. Implement `OnEditChangetext` as follows:

   ```
     void CClikDoc::OnEditChangetext()
     {
         CChangeText dlg;
   ▶     dlg.m_str = m_str;
   ▶     if (dlg.DoModal())
   ▶     {
   ▶         m_str = dlg.m_str;
   ▶         Refresh();
   ▶     }
     }
   ```

8. Add the following **#include** statement in AUTOCDOC.CPP:

   ```
   ▶ #include "dialogs.h"
   ```

Building and Running Autoclik Step 1

If you build and run Autoclik Step 1 now, it will only run 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 22 and 23.

CHAPTER 22

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 `CClikDoc` member variable `m_pt` by using the Get and Set methods in Autoclik's document dispatch interface.
- Expose the `CClikDoc` 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 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 `CClikDoc` if it is not already selected.
4. Choose the Add Property button.
5. In the Add Property dialog box, type **x** as the External Name.

   This is the name used by automation clients, as in the following Visual Basic code:

   ```
   Dim clik as object
   ...
   Set clik = CreateObject("Autoclik.Document")
   ...

   Sub SetX_Click()
       clik.X = X.Text
   End Sub
   ```

6. Under Implementation, select Get/Set Methods.

   You will use the other choice, Member Variable, for `m_str`.
7. In the Type box, select **short**.

8. Choose 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. Implement the Get and Set methods as indicated by the marked lines:

   ```
     short CClikDoc::GetX()
     {
   ▶     return (short)m_pt.x;
     }

     void CClikDoc::SetX(short nNewValue)
     {
   ▶     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.

10. Repeat steps 1 through 9 for the `y` property, ending with:

    ```
      short CClikDoc::GetY()
      {
    ▶     return (short)m_pt.y;
      }

      void CClikDoc::SetY(short nNewValue)
      {
    ▶     m_pt.y = nNewValue;
    ▶     Refresh();
      }
    ```

    Notice that ClassWizard allows you to implement methods the same way you implement member functions.

Look in AUTOCDOC.CPP to see how ClassWizard updated the dispatch map of the document class:

```
BEGIN_DISPATCH_MAP(CClikDoc, CDocument)
    //{{AFX_DISPATCH_MAP(CClikDoc)
    DISP_PROPERTY_EX(CClikDoc, "x", GetX, SetX, VT_I2)
    DISP_PROPERTY_EX(CClikDoc, "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 `CClikDoc` if it is not already selected.
4. Choose Add Property.
5. In the Add Property dialog box, type **text** in the External Name box.

   This is the name that the automation client uses to refer to the property.
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 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 `OnTextChanged` function by calling `Refresh( )`, just as you did for `SetX( )` and `SetY( )`. If you do not implement a similar `OnTextChanged` 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. Choose 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. Choose OK.
12. Open AUTOCDOC.H.

    ClassWizard has declared the following members in the dispatch map:

    ```
    //{{AFX_DISPATCH(CClikDoc)
    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 original `m_str` declaration, as indicated by the marked line:

    ```
      // Attributes
      public:
          CPoint m_pt;
    ▶     // CString m_str;  moved to dispatch map
    ```

14. Change the declaration of the dispatch map from **protected** to **public**, as indicated by the marked line:

    ```
    protected:
        //{{AFX_MSG(CClikDoc)
        afx_msg void OnEditChangeText();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

        // Generated OLE dispatch map functions
    ```

```
▶ public:
      //{{AFX_DISPATCH(CClikDoc)
      afx_msg short GetX();
      afx_msg void SetX(short nx);
      afx_msg short GetY();
      afx_msg void SetY(short ny);
      CString m_str;
      //}}AFX_DISPATCH
```

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 `CClikDocument`.

▶ **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 `CClikDoc` if it is not already selected.
4. Choose Add Method.
5. In the Add Method dialog box, type **RefreshWindow** in the External Name box.

   This is the name that the automation client uses to refer to the method, as in the following Visual Basic code:

   ```
   Sub RefreshDisplay_Click()
       clik.RefreshWindow
   End Sub
   ```

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, select **void**.
8. Choose 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 AUTOCDOC.CPP:

   ```
   void CClikDoc::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 AUTOCDOC.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 implementation of `Refresh` at the end of AUTOCDOC.CPP and the redundant declaration in AUTOCDOC.H:

    ```
    // Operations
    public:
    ▶ // Remove void Refresh ( );
    ```

    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 `CClikDoc` in the Class Name box, if it is not already selected.
4. Choose the Add Method button, which brings up the Add Method dialog.

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. Select **void** in the Return Type box.
7. Click in the Parameter 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.
   - Click under the Type heading, and select **short** as the Type.
8. Repeat step 7 for the y parameter.
9. Add the third parameter, `text`, selecting **LPCTSTR** from the Type.
10. Choose OK.

    This returns you to the Add Method dialog box, which shows the following implementation:

    ```
    void SetAllProps( short x, short y, LPCTSTR text);
    ```

11. Choose the Edit Code button.

    This takes you to the stub that ClassWizard created in AUTOCDOC.CPP:

    ```
    void CClikDoc::SetAllProps(short x, short y, LPCTSTR text)
    {
        // TODO:  Add your dispatch handler code here
    }
    ```

12. Replace the stub implementation with the marked lines:

    ```
      void CClikDoc::SetAllProps(short x, short y, LPCTSTR text)
      {
    ▶     m_pt.x = x;
    ▶     m_pt.y = y;
    ▶     m_str = text;
    ▶     Refresh();
      }
    ```

Take a look at the dispatch map for the `SetAllProps` method:

```
BEGIN_DISPATCH_MAP(CClikDoc, CDocument)
    //{{AFX_DISPATCH_MAP(CClikDoc)
    ...
    DISP_FUNCTION(CClikDoc, "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 must 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 allows the automation client to control 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 CClikApp::InitInstance()
{
    ...

    // Parse the command line to see if launched as OLE server
    if (RunEmbedded() || RunAutomated())
    {
        // Application was run with /Embedding or
        // /Automation. Don't show the
        //  main window in this case.
        return TRUE;
    }

    ...

    m_pMainWnd->DragAcceptFiles();
    // The main window has been initialized, so show and update it.
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();

    return TRUE;
}
```

▶ **To add a ShowWindow method**

1. Open ClassWizard.
2. Choose the OLE Automation tab.
3. In the Class Name box, select `CClikDoc` if it is not already selected.

4. Choose Add Method.

   This brings up the Add Method dialog box.
5. 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.
6. In the Return Type box, select **void**.
7. Choose OK.

   This returns you to the OLE Automation tab because this method has no parameters.
8. Choose the Edit Code button.

   This takes you to the stub implementation of `ShowWindow` created in AUTOCDOC.CPP by ClassWizard.
9. Implement `ShowWindow` as indicated by the marked lines:

```
  void CClikDoc::ShowWindow()
  {
▶     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);
▶     }
  }
```

Build and Test Autoclik Step 2

Autoclik is mostly implemented now. You can try it out with the Autodriv application, using the Disptest tool or Visual Basic. For more information on loading and using Autodriv and Disptest, see "Preview of the Autoclik Application" in Chapter 20.

Notice the difference between using Autodriv's `Set Text` versus `Set X` and `Set Y` functions. `Set X` and `Set Y` call Autoclik's `SetX` and `SetY` methods, which call `CClikDoc::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 `CClikDoc::Refresh`.

Note that the `GetPosition` and `SetPosition` methods have not been implemented. If you try to use Autodriv's `Get Position` or `Set Position` functions, Disptest or Visual Basic will return the message:

```
Ole Automation no such property or method
```

and put the focus on the offending Basic code in the Autodriv application.

CHAPTER 23

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 `CClikPoint`, 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 `CClikPoint`, 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. `CClikPoint`'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 principle 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. Choose the Add Class button.

   The Add Class dialog box appears.
3. In the Class Name box, type **CClikPoint**.
4. In the Header File box, change the filename to **autocpnt.h**.
5. In the Implementation File box, change the filename to **autocpnt.cpp**.
6. In the Class Type, select **CCmdTarget**.

   The OLE Automation and OLE Createable options now appear in the Add Class dialog box, because these options are only for **CCmdTarget**-derived classes.
7. Select the OLE Automation option.

   This enables the OLE Createable check box but doesn't select it.
8. Leave the OLE Createable option unselected and the External Name blank.

   This option is explained later.
9. Choose the Create Class button.

   This brings up the OLE Automation tab—the same tab you used to add properties and methods to Autoclik's document class.
10. Choose the Add Property button.
11. 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 `CClikPoint` dispatch interface class do not need notification functions.
    - Use the default Implementation type, Member Variable.
12. Choose OK.
13. Repeat steps 10 to 12 for the `y` property.
14. Choose OK.

Take a look at the CClikPoint class created by ClassWizard in AUTOCPNT.H and AUTOCPNT.CPP. The header file declares the dispatch map:

```
DECLARE_DISPATCH_MAP()
```

The AUTOCPNT.CPP file implements the dispatch map, reflecting the two properties you added in ClassWizard, x and y:

```
BEGIN_DISPATCH_MAP(CClikPoint, CCmdTarget)
    //{{AFX_DISPATCH_MAP(CClikPoint)
    DISP_PROPERTY(CClikPoint, "x", m_x, VT_I2)
    DISP_PROPERTY(CClikPoint, "y", m_y, VT_I2)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

Referring to One Dispatch Interface from Another

You will now add a property in Autoclik's document dispatch interface to expose the document's m_pt by using the second dispatch interface implemented by CClikPoint. You will expose this new property with a pair of Get and Set member functions, namely GetPosition and SetPosition. 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 Autodriv code, written in Visual Basic, accesses the Position property of Autoclik's document dispatch interface using the GetPosition property:

```
Dim clik As object
Dim pos As object

Sub Form_Load()
    Set clik = CreateObject("Autoclik.Document");
    clik.ShowWindow
End Sub

Sub GetPosition_Click()
    Set pos = clik.position
    X.Text = pos.X
    Y.Text = pos.Y
End Sub

Sub SetPosition_Click()
    Set pos = clik.position
    pos.X = X.Text
    pos.Y = Y.Text
    Set clik.position = pos
End Sub
```

In Visual Basic, Autoclik's `Position` property is declared simply as a generic "object." The code

```
Set pos = clik.position
```

accesses Autoclik's `Position` property, which is declared as **LPDISPATCH** in the MFC OLE automation server's dispatch map. The "clik" object is also declared simply as a generic "object" in this Visual Basic automation client application. The difference is that the automation client gets initial access to Autoclik's document dispatch interface object by creating it:

```
Set clik = CreateObject("Autoclik.Document");
```

whereas in the case of the `CClikPoint` object, the automation client obtains a reference by accessing the `CClikPoint` object as the "position" property of the "clik" dispatch interface object.

▶ **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 `CClikDoc` if it is not already selected.
4. Choose the Add Property button.
5. In the External Name box, type **Position**.
6. Under Implementation, choose Get/Set Methods.

   You cannot simply declare `Position` as a Member Variable. The additional work you need to do to allow the automation client to access this property is discussed in later steps.
7. Select **LPDISPATCH** in the Type box.
8. Accept the proposed member function names: `GetPosition` and `SetPosition`.
9. Choose OK.

   This returns you to the OLE Automation tab where the new property, `Position`, is shown with the following implementation:

   ```
   LPDISPATCH GetPosition( );
   void SetPositionLPDISPATCH newValue);
   ```
10. Choose the Edit Code button to implement the Get and Set member functions.

    This takes you to their stub implementations in AUTOCDOC.CPP.

11. Implement the Get and Set member functions as shown by the marked lines:

```
  LPDISPATCH CClikDoc::GetPosition()
  {
▶     CClikPoint* pPos = new CClikPoint;
▶     pPos->m_x = m_pt.x;
▶     pPos->m_y = m_pt.y;

▶     LPDISPATCH lpResult = pPos->GetIDispatch(FALSE);
▶     return lpResult;
  }

  void CClikDoc::SetPosition(LPDISPATCH newValue)
  {
▶     CClikPoint* pPos =
▶             (CClikPoint*)CCmdTarget::FromIDispatch(newValue);
▶     if (pPos != NULL && pPos->IsKindOf(RUNTIME_CLASS(CClikPoint)))
▶     {
▶         m_pt.x = pPos->m_x;
▶         m_pt.y = pPos->m_y;
▶         Refresh();
▶     }
  }
```

The implementation of `GetPosition` creates a new `CClikPoint` object. The `CClikPoint` 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 `CClikPoint` 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 `CClikPoint` object was constructed.

The implementation of `SetPosition` does a C++ down-casting of the **IDispatch** pointer to a `CClikPoint` pointer. It tests the down-casting with **IsKindOf** to make sure the automation client passed back an **IDispatch** pointer to a `CClikPoint` 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 `CClikPoint` class.

12. Add the following **#include** statement at the top of AUTOCDOC.CPP:

```
▶ #include "autocpnt.h"
```

This is required because the implementation of `SetPosition` refers to the `CClikPoint` class.

13. Change the declaration of the CClikPoint constructor from **protected** to **public** in AUTOCPNT.H:

```
▶ public:
      CClikPoint();
```

This is required because CClikDoc constructs the CClikPoint object in its implementation of GetPosition.

14. Change the declaration of CClikPoint's dispatch map from **protected** to **public**:

```
▶ public:
      // Generated OLE dispatch map functions
      //{{AFX_DISPATCH(CClikPoint)
      short m_x;
      short m_y;
      //}}AFX_DISPATCH
      DECLARE_DISPATCH_MAP()
```

This is required because CClikDoc directly accessed CClikPoint's member variables m_x and m_y in its implementation of GetPosition and SetPosition.

Createable OLE Dispatch Interface Objects

Earlier you were told not to choose the OLE Createable option for CClikPoint 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, a Visual Basic automation client could dynamically create a CClikPoint object, using

```
Dim pos As object

Set pos = CreateObject("Autoclik.Point");
```

Build and Run

Build the Step 3 version of Autoclik. It now performs exactly as you previewed it in Chapter 20. You have finished the Autoclik tutorial.

CHAPTER 24

Creating a Database Application

This tutorial shows you how to develop a Microsoft Foundation Class Library (MFC) database application. You'll learn how to:

- Use AppWizard and ClassWizard for database support.
- 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 6 through 15 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 that you will register with ODBC as the "Student Registration" data source name. Table 24.1 lists the tables, what they store, and the columns in them.

Table 24.1 Tables in the Student Registration Database

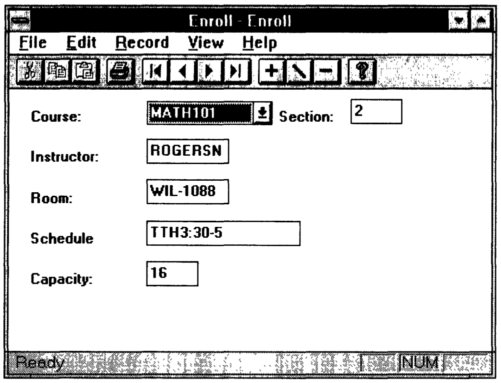
Table Name	Contents	Column List
Course	Think of each record as an entry in a course catalog. Example: the MATH101 course.	CourseID* CourseTitle Hours
Section†	A section record is a specific offering of a course at a specific time. For example, MATH101 may have many sections.	SectionNo* CourseID* InstructorID 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.

STDREG.MDB is located in the \MSVC20\SAMPLES\MFC\ENROLL directory. You can examine it, add records, and so on, using Microsoft Access.

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 24.1 shows what the Enroll application looks like at the end of the tutorial.



Course: MATH101 Section: 2

Instructor: ROGERSN

Room: WIL-1088

Schedule TTH3:30-5

Capacity: 16

Figure 24.1 The Enroll Tutorial Application

Setting Up the Student Registration Data Source

Before you start the tutorial, you need to set up a Student Registration database and register it as an ODBC data source.

Important To use the MFC database classes for targeting a Win32 platform (such as Windows NT) you must have the 32-bit ODBC driver for your data source. Drivers can be obtained from Microsoft and other vendors. Visual C++ provides a coupon for ordering the Microsoft ODBC Driver Fulfillment Kit (32-Bit). You should contact other driver vendors directly.

▶ To set up the student registration database

1. Choose a database format and ODBC driver for the Student Registration database.

   The Microsoft Foundation Classes support for database relies on Open Database Connectivity (ODBC). Choose a database format for which you have the corresponding database management system (DBMS) and a 32-bit ODBC driver.

   - Visual C++ 2.0 includes only the 32-bit ODBC driver for SQL Server. If you want to use MFC database support for SQL Server, you will need the SQL Server product in addition to the ODBC driver for SQL Server that is provided with Visual C++ 2.0.
   - If you want to use other database formats, you will need the DBMS as well as the ODBC driver. To obtain 32-bit ODBC drivers for other databases, contact Microsoft or other ODBC driver vendors directly.

- If you want to use Microsoft Access database format, you will need only the 32-bit Microsoft Access ODBC driver to create a database schema. This is an exception to the requirements listed above; however, if you intend to use the Microsoft Access database format, we recommend that you also use the Microsoft Access product itself in conjunction with MFC database support. The 32-bit ODBC driver for Microsoft Access is not included in Visual C++ 2.0. To obtain the Microsoft Access driver, contact Microsoft directly.

2. Install the 32-bit ODBC driver for your DBMS.

   You install the ODBC driver for your DBMS once, but you can use it with more than one data source. If you chose MFC Database Support during Setup, the 32-bit SQL Server driver is already installed. To install other drivers, open the ODBC Control Panel application and choose Drivers from the Data Sources dialog box. In the Drivers dialog box, choose Add. In the Add Driver dialog box, supply the path to the ODBC driver.

3. Create a new database by doing one of the following:

   - 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. You will use the STDREG tool to add tables to the new database in Step 5 of this procedure.

   –Or–

   - Copy the STDREG.MDB file from the \MSVC20\SAMPLES\MFC\STDREG sample directory on the Visual C++ CD to the corresponding sample directory on your hard drive. This Microsoft Access database file already contains tables and records used in the tutorial. If you use this file, continue with Step 4, but skip Step 5.

4. Register the new database with ODBC.

   You must 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 STDREG.MDB Microsoft Access database file.

   Register the Student Registration data source in one of the following ways:

   - Open the ODBC Control Panel tool and choose Add. The Add Data Source dialog box appears. For instructions on adding an ODBC data source using this dialog, see the encyclopedia article "ODBC Administrator" in *Programming with the Microsoft Foundation Class Library.*

–Or–

- Run the STDREG tool located in the \MSVC20\SAMPLES\MFC\BIN directory on the Visual C++ CD, and choose Add Data Source. The Add Data Source dialog appears. Use the instructions in the encyclopedia article "ODBC Administrator" in *Programming with the Microsoft Foundation Class Library* to add the data source. Go on to Step 5 without exiting the STDREG tool.

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 the ODBC Control Panel application to register your data sources.

5. If you are using a DBMS other than the prebuilt Microsoft Access STDREG.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 24.1. The tool also adds records to the newly created tables for use as test data by the Enroll application. The STDREG tool is located in the \MSVC20\SAMPLES\MFC\BIN directory on the Visual C++ CD, and the sources for the tool are located in the \MSVC20\SAMPLES\MFC\STDREG directory.

   Note The STDREG sources illustrate how to directly send SQL statements such as **CREATE TABLE**, and how to use ODBC catalog functions such as **SQLGetTypeInfo**.

   The STDREG tool displays the dialog box shown in Figure 24.2:

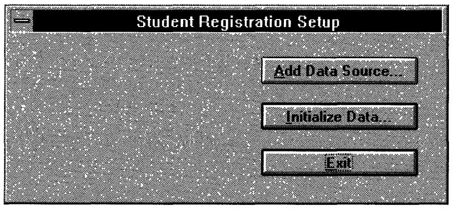
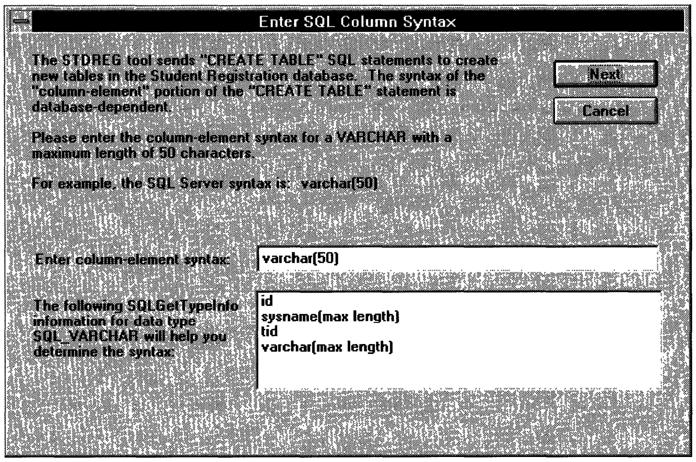


Figure 24.2 The STDREG Tool

Once you have registered the Student Registration data source, 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.

After logging in to the Student Registration data source, respond to a series of three Enter SQL Column Syntax dialog boxes, such as the one shown in Figure 24.3.



The STDREG tool sends "CREATE TABLE" SQL statements to create new tables in the Student Registration database. The syntax of the "column-element" portion of the "CREATE TABLE" statement is database-dependent.

Please enter the column-element syntax for a VARCHAR with a maximum length of 50 characters.

For example, the SQL Server syntax is: varchar(50)

Enter column-element syntax: varchar(50)

The following SQLGetTypeInfo information for data type SQL_VARCHAR will help you determine the syntax:

```
id
sysname(max length)
tid
varchar(max length)
```

Figure 24.3 The Enter SQL Syntax dialog box

Any given DBMS may define internal data types with names other than the standard data type names defined by ODBC. Normally, ODBC and MFC database applications do not need to refer to the data type names used internally by the DBMS; however the SQL **CREATE TABLE** statement is an important exception. ODBC does not attempt to interpret the data types specified for the one or more table columns specified in the **CREATE TABLE** statement. The application sending the **CREATE TABLE** statement must know the data type names supported by the specific DBMS.

The STDREG tool is designed to create tables in any arbitrary ODBC database. Therefore, it must determine how three ODBC data types used in the Student Registration application (**SQL VARCHAR**, **SQL INTEGER**, and **SQL SMALLINT**) are named internally by the DBMS. The STDREG tool queries the ODBC driver (using **SQLGetTypeInfo**) to find out what internal data types correspond to these SQL data types.

The ODBC driver may list more than one internal data type, or the driver may list data type creation parameters, such as "max length" for a VARCHAR. In these cases, it is difficult for STDREG, or any database-independent application, to choose the right internal data type, or to interpret the data type creation parameter. Therefore, STDREG displays the internal data types, as illustrated in the above dialog, and asks you to use this information to specify the correct syntax for the internal data type.

After you respond to three successive Enter SQL Column Syntax dialog boxes, STDREG creates the tables in the new database. 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.

Tutorial Steps

The tutorial consists of three steps. The following table describes the steps briefly.

Tutorial Step	Chapter	Description
1	25	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	26	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	27	Implement a user interface for adding, updating, and deleting records. Handle database exceptions.

The ENROLL sample program directory contains a subdirectory for each step, named STEP1, STEP2, and STEP3. Each step's subdirectory contains a Visual C++ project file, complete source files, and other files needed for the step.

Note The directory \ENROLL\STEP4 contains a fourth step, not covered in the tutorial. Step 4 illustrates additional class library database programming techniques, summarized below. See MFC Samples Help 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 25

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 25.1.

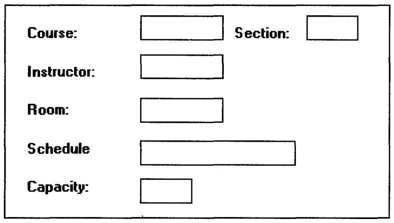


Figure 25.1 Enroll's Section Form

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 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" in *Programming with the Microsoft Foundation Class Library.*

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," "Record Views," "AppWizard: Database Support," "Data Source," and "ODBC" in *Programming with the Microsoft Foundation Class Library.*

Creating a New Database Application

For more information on how to use the Database Options button in AppWizard when you're creating your starter application, see "AppWizard: Database Support" in *Programming with the Microsoft Foundation Class Library.* 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 Support, No 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 the Microsoft Foundation Class Library*

The following procedure describes the steps for creating the Enroll application. For a more general and detailed procedure, see "AppWizard: Database Support" in *Programming with the Microsoft Foundation Class Library.*

▶ **To create the tutorial database application**

1. From the File menu, choose New.

   The New dialog box appears.
2. Select Project.

   The New Project dialog box appears.
3. In the Project Name box, type **enroll.**

   The application's project file will be given this name: in this case, ENROLL.MAK.
4. In the New Subdirectory box, delete "enroll" and type **myenroll.**
5. Specify the path to the project's subdirectory.

   Use the list box provided to navigate through the directories on the selected drive. As you navigate through the directory structure, the path listed in the dialog box changes to show where the named subdirectory (MYENROLL) should be placed. Navigate to \SAMPLES\MFC\ENROLL (relative to your Visual C++ installation). Assuming your Visual C++ installation is in directory MSVC20 on drive C, the path should look like this in the dialog box:

   ```
   c:\msvc20\samples\mfc\enroll\myenroll\enroll.mak
   ```
6. Choose the Create button.

   The MFC AppWizard Step 1 of 6 dialog box appears.
7. Choose the Next button in the AppWizard Step 1 dialog box to accept the default options.

   For more information on the various options that appear in this dialog boxes, see Chapter 1, "Creating a New Application Using AppWizard," in the *Visual C++ User's Guide.*
8. In the AppWizard Step 2 dialog box:
   - Select the "A database view, without file support" option.
   - Choose the Data Source button.

     The Data Sources dialog box appears.
   - Double-click "Student Registration" to select the data source. Depending on the database type, you may need to supply additional information to log into the data source.

The Select a Table dialog box appears.

- Double-click the table name SECTION. Depending on the data source type you are using, additional qualifiers may precede or follow the table name.
- Choose Next.

9. In the AppWizard dialog boxes for Steps 3, 4, and 5, choose Next 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.*

10. In the AppWizard Step 6 dialog box, check and modify class names and filenames that AppWizard suggests. To edit the information for a class, select the class name in the box at the top of the dialog box.

    - Select the class `CEnrollSet`, 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 **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 **sectform.h**. Change the implementation file name to **sectform.cpp**.

      The base class is **CRecordView**.

    When you complete the last dialog box, AppWizard will create a pair of classes derived from **CRecordset** and **CRecordView**: `CEnrollSet` and `CEnrollView`. By default, AppWizard bases the names of these 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. You'll change the names for the tutorial, even though Enroll uses only one recordset/view pair, as shown in the next procedure.

11. Choose Finish.
12. Choose OK in the New Project Information dialog box to accept the choices you've made.

AppWizard creates application, document, and frame window classes as usual. In addition, it creates `CSectionForm` (the record view class) and `CSectionSet` (the recordset class). `CSectionForm` is implemented in SECTFORM.H and SECTFORM.CPP. `CSectionSet` is implemented in SECTSET.H and SECTSET.CPP.

For more information about the AppWizard steps, see "AppWizard: Database Support" in *Programming with the Microsoft Foundation Class Library.*

Examining the Step 1 Classes

AppWizard makes ENROLL the current project.

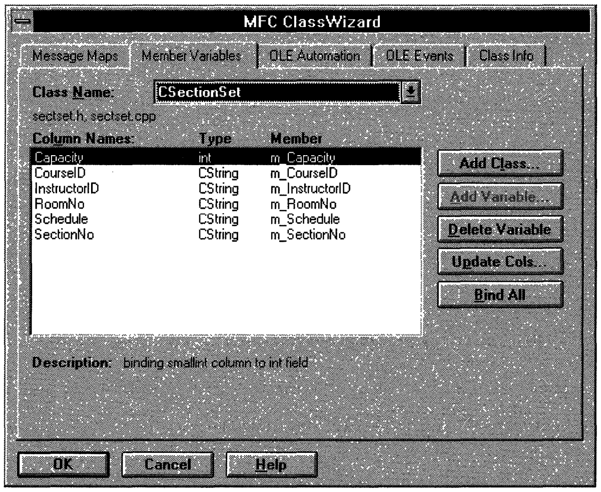
▶ **To examine the new recordset class with ClassWizard**

1. From the Project menu, choose ClassWizard.
2. Choose the Member Variables tab.
3. In the Class Name box, select `CSectionSet`.
4. Close ClassWizard when you finish the next section.

After examining `CSectionSet` with ClassWizard, you'll examine the source files for classes `CSectionForm` and `CEnrollDoc` using the text editor.

The CSectionSet Recordset Class

Figure 25.2 shows what you see in ClassWizard's Column Names box: a list of column names from the Section table. 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**.



Class Name: CSectionSet
sectset.h, sectset.cpp

Column Names:	Type	Member
Capacity	int	m_Capacity
CourseID	CString	m_CourseID
InstructorID	CString	m_InstructorID
RoomNo	CString	m_RoomNo
Schedule	CString	m_Schedule
SectionNo	CString	m_SectionNo

Description: binding smallint column to int field

Figure 25.2 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 the Microsoft Foundation Class Library*.

The CSectionForm Record View Class

To examine the source code for class `CSectionForm`, open file SECTFORM.CPP.

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 record view opens the recordset in its `OnInitialUpdate`:

```
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 in turn calling **CFormView::OnInitialUpdate**.

The CEnrollDoc Document Class

Notice that AppWizard creates a class derived from **CDocument**. To see the document class, open file ENROLDOC.H.

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 reveals that the role of the document class in Enroll is to own the recordset.

```
class CEnroll : 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 Support, No 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.

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 the Microsoft Foundation Class Library.*

For more information about documents and views, see Chapter 3 and Chapter 4 in *Programming with the Microsoft Foundation Class Library.*

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." Your task is to 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 the project window, double-click the ENROLL.RC resource file to open the resource browser. This browser displays the resources associated with a project.
2. In the resource browser, double-click Dialog.
3. Double-click IDD_ENROLL_FORM.

   The dialog editor opens and displays the dialog box with this ID.

   For more information about the dialog editor, see Chapter 5, "Using the Dialog Editor," in the *Visual C++ User's Guide.*
4. Select then delete the static control that says "TODO: Place form controls on this dialog."
5. Design Enroll's Section form to resemble Figure 25.3, using static controls and edit controls. Add the controls in pairs, in the following order: static text control, then corresponding edit control, and so on.

   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.

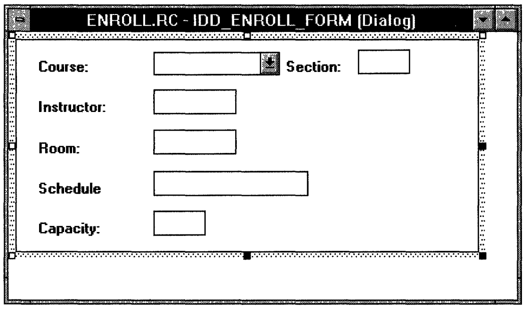


Figure 25.3 The Layout of Enroll's Section Form

6. Make the Course and Section edit controls read-only. For each of these two edit controls, 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.

7. Choose the Save command on the File menu.

   It's a good idea to periodically back up your work.

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 the Microsoft Foundation Class Library.*)

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`. This data member is a pointer to `CSectionSet`, Enroll's recordset class.

The control bindings go through `m_pSet` to the corresponding field data members of `CSectionSet`. For example, the Course edit control is bound to

```
m_pSet->m_CourseID
```

▶ **To bind a form control to a recordset data member**

1. In the resource browser, double-click Dialog.
2. Double-click **IDD_ENROLL_FORM** (if it isn't already open).

   The dialog editor opens and displays the dialog box with this ID.

   For more information about the dialog editor, see Chapter 5 in the *Visual C++ User's Guide.*
3. Hold down the CTRL key and double-click the Course edit control. ClassWizard's Add Member Variable dialog box appears. The most appropriate field name—based on the caption of the static control before the edit control in the tab order—is selected in the Member Variable Name box.

   For example, for `IDC_COURSE`, the control's caption is "Course," and the box should display:

   ```
   m_pSet->m_CourseID
   ```

The Member Variable Name box is a drop-down list box in which ClassWizard displays its best guess as to which recordset data member you want to map the selected control ID to. If this name is wrong, select another from the list.

4. Repeat step 2 for each of the other controls on the form.

   You can view the bindings by choosing the ClassWizard's Member Variables tab.

5. Choose OK to close ClassWizard.
6. Save your work.

Note Using CTRL+double-click in the dialog editor is a new 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.

Build and Run Enroll Step 1

Build and run Enroll Step 1. For information on building, see Chapter 2, "Working with Projects," in the *Visual C++ User's Guide.*

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. 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 data source when you move to another record. As mentioned earlier, the key fields Course and Section are read-only.
- When you finish, exit the program.

This completes Step 1 of the database tutorial. Chapter 26 continues by showing you how to add a second recordset and use it to fill a combo box control on the form.

CHAPTER 26

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. To view more than one recordset with the same record view, you'll need to add a little code in Step 2.

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. Lines of code that you should enter are shown marked in the margin with a ▸symbol. 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.

Try Step 2 out now, if you like. Load the ENROLL.MAK project from \ENROLL\STEP2, then build and run it. 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 You may observe that menu commands and toolbar buttons in Enroll aren't always enabled or disabled as you would expect. The `CSectionForm` record view is not able to detect the end of the recordset until the user has moved past it. If the user moves sequentially past the last record and then moves back to the last record (or before it), the record view can track the user's position in the recordset and disable user interface objects correctly. 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 need to change it to a drop-list style combo box. Continue using the same ENROLL.MAK you used for Enroll Tutorial Step 1. For details about how to perform the steps in the following procedure, see the *Visual C++ User's Guide*.

▶ **To change the Course control to a combo box**

1. Open the dialog resource whose ID is **IDD_ENROLL_FORM**.
2. Select then delete the Course edit control.
3. Add a combo box where the edit control was.

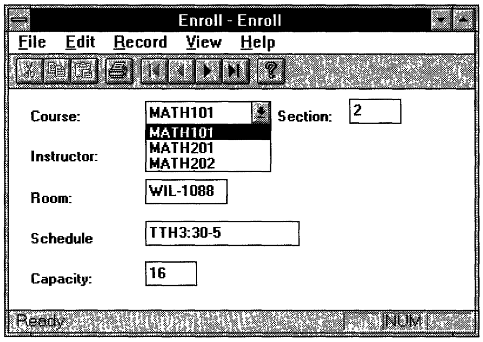
   The status bar lets you know if you've chosen the right control from the control palette.
4. Open the Properties window for the combo box control and specify the following:
   - In the ID box on the General property page, type **IDC_COURSELIST**.
   - In the Type box on the Styles property page, choose Drop List.
5. Increase the size of the combo 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.
6. Check that the new control follows its static text label in the tab order.

   If you do, ClassWizard can later use the label as the basis for presenting the recordset member to which you're most likely to want to bind the control. See the *Visual C++ User's Guide* for information about setting the tab order.

   Leave the dialog editor open.

Figure 26.1 shows the final appearance of Enroll Step 2 with the combo box in place.



Course: MATH101
MATH101
MATH201
MATH202
Section: 2
Instructor:
Room: WIL-1088
Schedule TTH3:30-5
Capacity: 16

Figure 26.1 Enroll Step 2 With a Combo Box

For general information about ClassWizard, see the *Visual C++ User's Guide.*

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 and bind the new combo box control to the CourseID field. In addition, you need to 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. 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 Project 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. Choose 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. This opens ClassWizard's Add Variable Member dialog box. Use the default Variable Type for a combo box control, which is **CString**.
2. Select `m_pSet->m_CourseID` from the Member Variable Name drop list.
3. Choose OK to close the dialog box.

▶ **To bind the combo box control to the view's CComboBox member variable**

1. Press CTRL and double-click the combo box control to open the Add Variable Member dialog box again.
2. In the Member Variable Name box, give the **CComboBox** member variable the name `m_ctlCourseList`.
3. In the Category box, select Control.
4. Close the dialog box.
5. Save your work.

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 Project menu, choose ClassWizard.
2. Choose Add Class to open the Add Class dialog box.
3. In the Class Name box, type **CCourseSet**
4. In the Class Type box, select **CRecordset**.
5. In the Header File box, change the name of the file to **coursset.h**.
6. In the Implementation file box, change the name of the file to **coursset.cpp**.
7. Choose the Create Class button, which opens the SQL Data Sources dialog box.

▶ **To connect the recordset class to the Course table**

1. In the SQL Data Sources dialog box, double-click the data source name "Student Registration." Depending on the database type, you may need to supply additional information to log in to the data source.

   The Tables dialog box opens.

2. In the Tables dialog box, double-click the table name "COURSE." 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 ClassWizard's Member Variables tab. The Class Name box shows `CCourseSet`, and three names are listed in the Column Names box.

   Table 26.1 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 variable. ClassWizard lets you delete those variables if you don't need to access or modify the columns—but you must be careful not to delete a field member variable for a column that is part of the table's primary key.

Table 26.1 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 the Microsoft Foundation Class Library.*

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. Open file ENROLDOC.H.
2. Declare an embedded `CCourseSet` object, as shown by the marked line in the following code:

```
class CEnrollDoc : public CDocument
{
...
// Attributes
public:
    CSectionSet m_sectionSet;
▶       CCourseSet  m_courseSet;
...
};
```

3. In the ENROLDOC.CPP, ENROLL.CPP, and SECTFORM.CPP implementation files, add a **#include** directive for "coursset.h" before the existing **#include** directive for "enroldoc.h", as shown in the following lines:

```
...
#include "sectset.h"
▶   #include "coursset.h"
#include "enroldoc.h"
...
```

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. Lines to add are marked with the ▸ symbol.

▶ **To fill the combo box**

1. Open the file SECTFORM.CPP.
2. In the implementation of `OnInitialUpdate`, add the code marked below:

```
void CSectionForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_sectionSet;

▶       // 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;

    CRecordView::OnInitialUpdate();


▶       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);
}
```

4. Save your work.

For more information, see the article "Record Views: Filling a List Box from a Second Recordset" in *Programming with the Microsoft Foundation Class Library*.

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 illustration. Do not add any code to your source files in this section.

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 **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.

▶ **To declare a parameter data member in the recordset's header file**

1. Open file SECTSET.H.
2. Add the following member variable declaration for `m_strCourseIDParam`:

```
public:
...
//{{AFX_FIELD(CSectionSet, CRecordset)
...
//}}AFX_FIELD
▶    CString m_strCourseIDParam;
```

▶ **To bind the parameter data member to the recordset**

1. Open file SECTSET.CPP.
2. In the CSectionSet constructor, initialize the parameter count variable, **m_nParams**, which by default is zero. Also initialize Enroll's single parameter, m_strCourseIDParam, as shown in the following marked code:

```
...
//{{AFX_FIELD_INIT(CSectionSet)
...
m_nFields = 6;
//}}AFX_FIELD_INIT
▶     m_nParams = 1;
▶     m_strCourseIDParam = "";
```

3. In the DoFieldExchange member function definition, add two lines of code to identify m_strCourseIDParam as a parameter data member.

```
void CSectionSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CSectionSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, "CourseID", m_CourseID);
    RFX_Text(pFX, "SectionNo", m_SectionNo);
    RFX_Text(pFX, "InstructorID", m_InstructorID);
    RFX_Text(pFX, "RoomNo", m_RoomNo);
    RFX_Text(pFX, "Schedule", m_Schedule);
    RFX_Int(pFX, "Capacity", m_Capacity);
    //}}AFX_FIELD_MAP
▶       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.

4. 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 This section adds no new code to Enroll.

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 the *Microsoft Foundation Class Library Samples* under Microsoft Foundation Class Library in the Contents browser.

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 the Microsoft Foundation Class Library.*

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.

▶ **To requery the CSectionSet recordset**

1. From the Project menu, choose ClassWizard.
2. Choose the Message Maps tab.
3. In the Class Name box, select `CSectionForm`.
4. In the Object IDs box, select `IDC_COURSELIST`.
5. In the Messages box, select **CBN_SELENDOK**.
6. Choose Add Function to open the Add Member Function dialog box.
7. Choose OK to accept the default name `OnSelendokCourseList`.

   You can change the name if you prefer.
8. Choose Edit Code to jump to the code in the source editor.

9. Add the marked lines of code in the following:

```
   void CSectionForm::OnSelendokCourselist()
   {
▶          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);
   }
```

10. 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" in *Programming with the Microsoft Foundation Class Library.*

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 27 (Step 3) concludes the tutorial by showing you how to add and delete records.

CHAPTER 27

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. Lines of code that you should enter are shown marked in the margin with a ▸ symbol. 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 the Microsoft Foundation Class Library (MFC).

The user interface in Step 3 includes three new commands on the Record menu, with corresponding toolbar buttons:

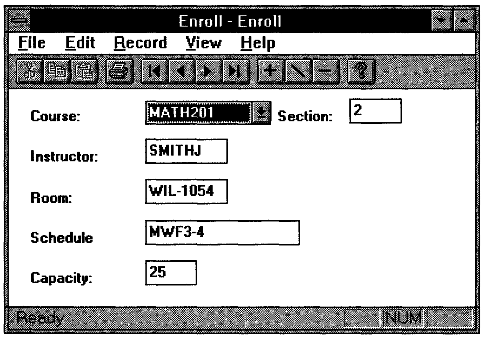
- The Add command prepares a blank record into which the user enters data. The user saves the new record by moving to another record, just as he or she saves an edited record by moving to another record. The user can also save the new record by issuing the Add command again.
- The Refresh command abandons 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. Load the ENROLL.MAK project from \ENROLL\STEP3, then build and run it.

Here are some things to try:

- Try the new Add, Refresh, and Delete commands.
- Try forcing the two exceptions handled by Enroll:
  1. Try to delete a section that has Enrollment records.
  2. Try to add a duplicate section.

When you finish, exit the program. Figure 27.1 shows the finished Enroll application.



Course: MATH201 Section: 2
Instructor: SMITHJ
Room: WIL-1054
Schedule MWF3-4
Capacity: 25

Figure 27.1 The Enroll Step 3 Application

Creating the Step 3 User Interface

The first thing to do in Step 3 is to use the menu and the accelerator editors to add menu items and an accelerator for the new commands. You also need to add message handler member functions for these commands.

Add Menu Items for Add, Refresh, and Delete

▶ **To add menu items for the commands**

1. In the resource browser, double-click Menu.
2. Double-click the **IDR_MAINFRAME** menu resource
3. Open its Record menu.
4. At the top of the Record menu, add an "Add" menu item with the following caption, resource ID, and command prompt:
   - **&Add**
   - **ID_RECORD_ADD**
   - **"Add a new section."**
5. 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 the key that will be used as an accelerator.
6. Add a "Delete" menu item with the following caption, resource ID, and command prompt:
   - **&Delete**
   - **ID_RECORD_DELETE**
   - **"Delete section."**
7. Add a separator.
8. Save your work and leave the menu editor window open.

   You'll need the editor open to establish a context the next time you open ClassWizard.

The command IDs are application-specific IDs, not defined by the framework as are **ID_RECORD_FIRST** and the other commands on the Record menu.

Figure 27.2 shows the completed menu in the menu editor.

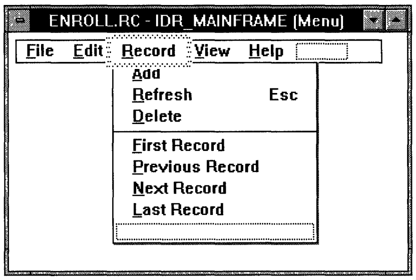


Figure 27.2 The Record Menu with New Commands

Add an Accelerator for the Refresh Command

You can skip this step if you wish, since you can test the application without this accelerator.

▶ **To add an accelerator**

1. In the resource browser, double-click Accelerator.
2. Double-click the **IDR_MAINFRAME** accelerator resource.
3. Create a new accelerator using the ESC key, with the following ID: **ID_RECORD_REFRESH**.

   VK_ESCAPE is defined as the accelerator for the Refresh command. Be sure to clear the "Ctrl" modifier box.

   For information about creating and editing accelerators, see Chapter 7, "Using the Accelerator Editor," in the *Visual C++ User's Guide*.
4. Save your work but leave the accelerator editor open.

   You'll need this editor or the menu editor open to establish a context the next time you open ClassWizard.

Create Handlers for Add, Refresh, and Delete

Each of the new commands needs a command handler function in the `CSectionForm` class.

▶ **To create handlers for the commands**

1. With the focus on the **IDR_MAINFRAME** menu or accelerator resource, open ClassWizard.
2. Choose the Message Maps tab.
3. In the Class Name box, select class `CSectionForm`.

4. Select the `ID_RECORD_ADD` command ID, select `COMMAND` in the Messages box, and choose Add Function to add a command handler function.

   Accept the default handler name: `OnRecordAdd`.
5. Repeat step 4 for the `ID_RECORD_DELETE` and `ID_RECORD_REFRESH` command IDs.
6. Choose OK to close ClassWizard.

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:
  1. Prepares the current record for updating by calling the recordset's **Edit** member function.
  2. It 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.
  3. 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:
  1. 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++.)
  2. Change the recordset's member variables, usually by getting the new values from the form's controls with **UpdateData**.
  3. 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 procedure implements the add mode and creates a **CEdit** member variable used to turn on and off the read-only style of the Section edit control.

▶ **To prepare for implementing the Add command**

1. Add the `m_bAddMode` data member to `CSectionForm` in file SECTFORM.H:

```
   // Attributes
   public:
       CMyenrollDoc* GetDocument();
 ▶ protected:
 ▶     BOOL m_bAddMode;
```

2. Initialize `m_bAddMode` in the `CSectionForm` constructor in file SECTFORM.CPP:

```
   CSectionForm::CSectionForm()
       : CRecordView(CSectionForm::IDD)
   {
       //{{AFX_DATA_INIT(CSectionForm)
       m_pSet = NULL;
       //}}AFX_DATA_INIT
 ▶     m_bAddMode = FALSE;
   }
```

3. To define a **CEdit** member variable for the Section edit control in `CSectionForm`, open ClassWizard and choose the Member Variables tab.

   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.

4. In the Class Name box, select `CSectionForm`.
5. In the Control IDs box, select **IDC_SECTION**, which is already associated with a **CString** member.
6. Choose Add Variable to open the Add Member Variable dialog box.
7. In the Variable Name box, type the name **m_ctlSection**
8. In the Property box, select **Control**.

   Notice that the Variable Type box changes appropriately to **CEdit**.

9. 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`.

10. 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**

- Implement the `OnRecordAdd` handler that ClassWizard created by adding the following code in file SECTFORM.CPP:

```
  void CSectionForm::OnRecordAdd()
  {
▶     // 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 **CRecordView**'s **OnMove** member function.

▶ **To implement Add functionality in the OnMove function override**

1. From the Project menu, run ClassWizard.
2. Select the Message Map tab.
3. In the Class Name box, select class `CSectionForm`.
4. In the Object ID box, select **CSectionForm**, then select **OnMove** in the Message box.

5. Select Add Function. ClassWizard automatically highlights the **OnMove** entry in the Member Functions box.
6. Select Edit Code.
7. Replace the stub `OnMove` function with the following code:

```
  BOOL CSectionForm::OnMove(UINT nIDMoveCommand)
  {
▶     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" in *Programming with the Microsoft Foundation Class Library.*

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**

- Place an **if** block around the requery code in the `OnSelendokCourseList` handler in file SECTFORM.CPP, as shown by the marked lines in the following code:

```
  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**

- Add the marked lines to the `OnRecordDelete` handler in SECTFORM.CPP that ClassWizard created:

```
  void CSectionForm::OnRecordDelete()
  {
▶     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**

- Add the marked code below to the `OnRecordRefresh` handler function in file SECTFORM.CPP:

```
  void CSectionForm::OnRecordRefresh()
  {
▶     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 toolbar buttons for Enroll. For more information on creating toolbar buttons, see "Edit Scribble's Toolbar" in Chapter 10.

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.

The directory \ENROLL\STEP4 contains a fourth step, not covered in the tutorial. Step 4 illustrates additional class library database programming techniques, summarized below. See ENROLL in the *Microsoft Foundation Class Library Samples* under Microsoft Foundation Class Library in the Contents browser 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.

PART 3

Appendixes

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 products and services, which make Microsoft Windows and Microsoft Visual C++ more accessible for people with disabilities:

- Microsoft Visual C++ accessibility.
- Microsoft services for people who are deaf or hard-of-hearing.
- Access Pack for Microsoft Windows, a software utility that makes using Microsoft Windows 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 cassettes and floppy disks.
- Products for people who are blind or have low vision.
- Hints for customizing Microsoft Windows.
- Other products and services for people with disabilities.

Note The information in this section applies only to users who purchased Windows in the United States. If you purchased Windows outside the United States, your Windows 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 and Service on a text telephone by dialing (800) 892-5234 between 6:30 A.M. and 5:30 P.M. Pacific time. For technical assistance you can contact Microsoft Product Support Services on a text telephone at (206) 635-4948 between 6:00 A.M. and 6:00 P.M. Pacific time. Microsoft support services are subject to Microsoft prices, terms, and conditions in place at the time the service is used.

Access Pack for Microsoft Windows

Microsoft distributes Access Pack for Microsoft Windows, which provides people with motion or hearing disabilities better access to computers running Microsoft Windows. Access Pack for Microsoft Windows contains 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 ACCESS.EXE. If you have a modem, you can download Microsoft Windows Driver Library components from the following network services:

- CompuServe®
- GEnie™
- Microsoft OnLine

- Microsoft Download Service (MSDL), which you can reach by calling (206) 936-MSDL (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, or 9600
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)

People within the United States who do not have a modem can order the Access Pack for Microsoft Windows on disks by calling Microsoft Product Support Services at (206) 637-7098 or (206) 635-4948 (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 in order to use these features.

Microsoft Windows already supports 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, choose International from the Windows Control Panel. The two layouts for people who type with one hand are distributed as Microsoft Application Note GA0650. It is also contained in file GA0650.ZIP on most network services or GA0650.EXE on the Microsoft Download Service. For instructions on obtaining this application note see the preceding section, "Access Pack for Microsoft Windows."

Microsoft Documentation on Audio Cassettes and Floppy Disks

People who have difficulty reading or handling printed documentation can obtain most Microsoft publications from Recording for the Blind, Inc. Recording for the Blind distributes these documents to registered 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:

Recording for the Blind

Address	Phone	Phone outside U.S.	FAX
20 Roszel Road Princeton, NJ 08540	(800) 221-4792	(609) 452-0606	(609) 987-8116

Products for People Who Are Blind or Have Low Vision

There are numerous products available to help people who are blind or have low vision use Microsoft Windows. For people with low vision there are screen-enlargement utilities, and for people who cannot use visual information there are screen readers that provide alternative output by synthesized voice or refreshable Braille displays. In addition, people with low vision can customize the Microsoft Windows display to suit their needs.

For more information on the various products available, see "Getting More Information" later in this appendix. For more information about customizing Microsoft Windows for people with low vision, see the next section, "Customizing Windows."

Customizing Windows

There are many ways you can adjust the appearance and behavior of Microsoft Windows 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.

For information relating to customizing Windows 3.0 for people with disabilities, see Application Note WW0786; for Windows 3.1, Application Note WW0787; for Windows for Workgroups NT 3.1, Application Note WG0788; for Windows NT 3.1, Application Note WN0789. For information on obtaining application notes, see "Access Pack for Microsoft Windows," earlier in this appendix.

Getting More Information

For more information on Microsoft products and services for people with disabilities, contact Microsoft Sales and Service at (800) 426-9400 (voice) or (800) 892-5234 (text telephone).

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 17,000 products and other information for people with disabilities. It is issued twice a year.

You can contact the Trace R&D Center at the following address or telephone numbers:

Trace R&D Center

Address	Phone	Phone outside U.S.	FAX
S-151 Waisman Center 1500 Highland Avenue Madison, WI 53705-2280	(608) 263-2309	(608) 263-5408	(608) 262-8848

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

Address	Voice/text telephone outside South Carolina	Voice/text telephone inside South Carolina	Voice/text telephone outside the U.S.	FAX
Benson Building Univ. of South Carolina Columbia, SC 29208	800) 922-9234, ext. 301	(800) 922-1107	(803) 777-6222	(803) 777-6058

APPENDIX B

Microsoft Support Services

Visual C++ has many resources to help answer developer's questions. Sources of documentation include the online Help, Books Online, README.WRI, technical notes, and samples. Help is also available through the Microsoft Support Network, which provides technical support through a variety of services.

Troubleshooting Guide

Most problems can be solved using the information provided with Visual C++. Here are some steps which can help you take advantage of the Visual C++ resources and help you isolate the problem if you need to call technical support.

Check the Product Documentation

This is one of the most productive ways to find answers to questions, and it can save you time and money. You can consult several types of documentation:

- Books Online. By clicking the Books Online icon, you have access to over 5000 pages of Visual C++ documentation. You can scan for information in the Contents browser, use Search to search for a keyword, or use Search Plus to do a full-text search on a subject.
- Help. This includes procedural and reference information on common features, functions, and error messages. You can access Help through the Help menu, or by pressing F1 on a function in your source code.
- README.WRI. This file contains late-breaking information about configuration problems, new features, and known bugs. You can also open README.WRI by clicking the icon in the Visual C++ program group.
- Samples. Two online books, MFC Samples, and Samples, include sample programs that illustrate common programming tasks. You can open these by clicking the books online icon.
- Microsoft Knowledge Base. The Microsoft Knowledge Base contains thousands of articles on known problems and programming issues. It is available through the Help menu or on the Microsoft Developer's Network CD-ROM, and it can also be accessed from CompuServe or Internet.

Reproduce the Problem

Reproducing the problem is the first step in solving it. Once you can reproduce the problem, you can start finding solutions. The following questions may give you more insight on the problem:

- Does the problem occur with just this one program? You may want to try one of the samples to see if you can reproduce the problem with it. If you cannot reproduce the problem with other programs, think about what's specific about the program.
- Does the problem occur on just your machine? If so, the problem may be related to your system configuration. Try using a different Windows video driver or modify your system configuration to see if the problem still occurs. It's a good idea to try to make your machine as much like the average machine as possible.
- What versions of the tools are you using? Knowing the version of the compiler, linker, and other tools makes it easier to reproduce (or avoid) the problem in the future.
- Under what circumstances does the problem occur? Does it only occur when you build from the command line, or within Visual C++? Does the amount of available memory affect the problem? How about other programs that are running in the system?

Isolate the Problem

After seeing what circumstances cause the problem, you may be able to isolate it. Once a problem is isolated, it's much easier and quicker to fix or work around.

- Try isolating the component that's causing the problem. You can use the information about what conditions it reproduces to help isolate the component. For example, if a problem occurs when compiling both inside the development environment and using NMAKE from the command line, the problem probably isn't with the development environment.
- If the problem is with the compiler, you may be able to create a small example. The compiler generates code on a per-function basis, and you might be able to isolate it to a particular module or function. A useful way to comment out large blocks of code is to use `#if 0`.
- Sometimes you can isolate a problem by breaking things in half. For example, if a particular module is causing a LINK error, separating the module into two modules will help isolate the problem.

Product Support Within the United States and Canada

In the United States and Canada, the following support services are available through the Microsoft Support Network:

- Electronic services
- Standard support
- Priority support
- Text telephone
- Product training and consultation
- Other support options

Electronic Services

These services are available 24 hours a day, 7 days a week, including holidays.

Microsoft FastTips (800) 936-4300 on a touch-tone telephone. Receive automated answers to common questions and access a library of technical notes, all delivered by recording or fax. After you reach FastTips, use the following keys to move through the automated system:

- To advance to the next message, press the ASTERISK (*) key.
- To repeat the current message, press 7.
- To return to the beginning of FastTips, press the POUND SIGN (#) key.

CompuServe Interact with other users and Microsoft support engineers, or access the Microsoft Knowledge Base to get product information. At any ! prompt, type **go microsoft** to access all Microsoft forums, or type **go mskb** to access the Microsoft Knowledge Base. Type **go mslang** to access the Microsoft Languages forum and Visual C++ sections, or **go msmfc**, to access the Microsoft Foundation Class forum. For an introductory CompuServe membership kit specifically for Microsoft users, dial (800) 848-8199 and ask for operator 524.

Microsoft Download Service Access, via modem, the Driver Library and the most current technical notes (1200, 2400, or 9600 baud; no parity; 8 data bits; 1 stop bit.) In the United States, call (206) 936-6735. In Canada, call (905) 507-3022.

Internet Access the Driver Library and the Microsoft Knowledge Base. The Microsoft Internet FTP archive host, ftp.microsoft.com, supports anonymous login. When logging in as anonymous, you should type your complete electronic mail names as your password.

Standard Support

In the United States, 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. This support is available for 30 days after you make your first call.

- For technical support for Visual C++, call (206) 635-7007.

In Canada, support engineers are available via a toll call between 8:00 A.M. and 8:00 P.M. Eastern time, Monday through Friday, excluding holidays. Call (905) 568-3503. This support is available for 30 days after you make your first call.

When you call, you should be at your computer with Microsoft Visual C++ running and the product documentation at hand. Be prepared to give the following information:

- The version of Microsoft Visual C++ you are using.
- Your product identification number (choose About from the Help menu to find this).
- The type of hardware you are using, including network hardware, if applicable.
- The operating system you are using.
- The exact wording of any messages that appeared on your screen and the error number, if any.
- A description of what happened and what you were trying to do when the problem occurred.
- 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, except holidays.

- In the United States, call (900) 555-2300; $2 (U.S.) per minute; $95 (U.S.) maximum. Charges appear on your telephone bill.
- In the United States, call (800) 936-5800; $95 (U.S.) per incident, billed to your VISA card, 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 between 6:00 A.M. and 6:00 P.M. Pacific time, Monday through Friday, excluding holidays. In Canada, using a TT/TDD modem, dial (905) 568-9641 between 8:00 A.M. and 8:00 P.M. Eastern time, Monday through Friday, excluding holidays.

Other Support Options

The Microsoft Support Network offers annual and multiple incident support plans. For information, in the United States, contact the Microsoft Support Network Sales and Information group at (800) 936-3500 between 6:00 A.M. and 6:00 P.M. Pacific time, Monday through Friday, excluding holidays. In Canada, call (800) 668-7975 between 8:00 A.M. and 8:00 P.M. Eastern time, Monday through Friday, excluding holidays.

Product Training and Consultation

Microsoft Solution Providers are independent organizations that provide consulting, integration, customization, development, technical support and training, and other services for Microsoft products. These companies are called Solution Providers because they apply technology and provide services to help solve real-world problems.

In the United States, for more information about the Microsoft Solution Providers program or the Microsoft Solution Provider nearest to you, please call (800) 426-9400 between 6:30 A.M. and 5:30 P.M. Pacific time, Monday through Friday, excluding holidays. In Canada, call (800) 563-9048 between 8:00 A.M. and 8:00 P.M. Eastern time, Monday through Friday, excluding holidays.

Product Support Worldwide

If you are outside the United States and have a question about a Microsoft product, first:

- Consult the documentation and other printed information included with your product.
- Check online Help.

- Check the README files that come with your product disks. These files provide general information that became available after the books in the product package were published.
- Consult electronic options such as CompuServe forums or bulletin boards, if available.

If you cannot find a solution, you can receive information on how to obtain product support by contacting the Microsoft subsidiary office that serves your country.

The Microsoft Support Network

The Microsoft Support Network, where available, offers you a wide range of choices and access to high quality, responsive technical support. Microsoft recognizes that support needs vary from user to user; the Microsoft Support Network allows you to choose the type of support that best meets your needs, with options ranging from electronic bulletin boards to annual support programs.

The Microsoft Support Network is subject to Microsoft's then-current prices, terms, and conditions in place in each country at the time the services are used and is subject to change without notice.

Calling a Microsoft Subsidiary Office

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

Microsoft subsidiary offices and the countries they serve are listed below.

If there is no Microsoft office in your country, please contact the establishment from which you purchased your Microsoft product.

Microsoft Subsidiary Offices

Area	Telephone Numbers
Argentina	Microsoft de Argentina S.A. Customer Service: (54) (1) 814-5105 (54) (1) 814-4807 (54) (1) 814-4808 (54) (1) 811-7199 Fax: (54) (1) 814-0372 Technical Support: (54) (1) 815-1521
Australia	Microsoft Pty. Ltd. Fax: (61) (02) 805-0519 Sales Information Centre: (61) (02) 870-2100 Installation Support: (61) (02) 870-2870 Bulletin Board Service: (61) (02) 878-5200 Technical Support: (61) (02) 870-2131
Austria	Microsoft Ges.m.b.H. Phone: 0222-68 76 07 Fax: 0222-68 16 2710 Information: 0660-6520 Prices, updates, etc.: 0660-6520 CompuServe: GO MSEURO (Microsoft Central Europe) Technical support: C/C++, FORTRAN, Macro Assembler PDS: 0660-6515
Belgium	Microsoft NV Phone: 02-7303911 Customer Service: 02-7303922 CompuServe: 02-2150530 (GO MSBEN) Bulletin Board: 02-7350045 (1200/2400/9600 bd, 8N1, ANSI) Technical Support: 02-5133274 (Dutch speaking) 02-5023432 (English speaking) 02-5132268 (French speaking)
Bolivia	See Argentina
Brazil	Microsoft Informatica Ltda. Phone: (55) (11) 530-4455 Fax: (55) (11) 240-2205 Technical Support Phone: (55) (11) 871-0090 Technical Support Fax: (55) (11) 241-1157 Technical Support Bulletin Board Service: (55) (11) 872-4106

Area	Telephone Numbers
Canada	Microsoft Canada Inc. Head Office Phone: 1 (905) 568-0434 Customer Support Centre: 1 (800) 563-9048 Microsoft Support Network: Standard Technical Support Phone: 1 (905) 568-3503 Priority Support Information: 1 (800) 668-7975 Text Telephone (TT/TDD) 1 (905) 568-9641 Technical Support Bulletin Board Service: 1 (905) 507-3022
Caribbean	Microsoft Caribbean, Inc. Phone: (809) 273-3600 Fax: (809) 273-3636 Technical Support: (214) 714-9100
Chile	Microsoft Chile S.A. Phone: 56 2 218 5771 56 2 218 5711 56 2 218 7524 Fax: 56 2 218 5747
Colombia	Microsoft Colombia Phone: (571) 618 2245 Fax:(571) 618 2269 Technical Support: (571) 618 2255
Denmark	Microsoft Denmark AS Phone: (45) (44) 89 01 00 Microsoft Sales Support: (45) (44) 89 01 90 Microsoft FaxSvar: (45) (44) 89 01 44 Microsoft BBS: (45) (44) 66 90 46 (Document 303030 in FaxSvar contains detailed instructions) Technical Support: (45) (44) 89 01 11
Dubai	Microsoft Middle East Phone: (971) 4 513 888 Fax: (971) 4 527 444
England	see United Kingdom
Finland	Microsoft OY Phone: (358) (0)9 0 525 501 Microsoft FaxSvar: (46) (0)8 752 29 00 (Information in Swedish and English) Microsoft BBS: (46) (0) 8 750 47 42 (Information in Swedish and English) For Technical Support, please contact your local dealer.

Area	Telephone Numbers
France	Microsoft France Phone: (33) (1) 69-86-46-46 Fax: (33) (1) 64-46-06-60 Telex: MSPARIS 604322 Technical Support Phone: (33) (1) 69-86-10-20 Technical Support Fax: (33) (1) 69-28-00-28 Fax Information Service: (33) (1) 69-29-11-55
French Polynesia	See France
Germany	Microsoft GmbH Phone: 089-3176-0 Fax: 089-3176-1000 Telex: (17) 89 83 28 MS GMBH D Information: 089-3176 1199 Prices, updates, etc.: 089-3176 1199 CompuServe: GO MSEURO (Microsoft Central Europe) Bulletin board, device drivers, tech notes: Btx: microsoft# or *610808000# Technical support: C/C++, FORTRAN, Macro Assembler PDS: 089/3176-1150
Greece	Microsoft Hellas, S.A. Phone: (30) (1) 6893 631 through (30) 1 6893 635 Fax: (30) (1) 6893 636
Hong Kong	Microsoft Hong Kong Limited Fax: (852) 560-2217 Technical Support: (852) 804-4222
Ireland	See United Kingdom
Israel	Microsoft Israel Ltd. Phone: 972-3-575-7034 Fax: 972-3-575-7065
Italy	Microsoft SpA Phone: (39) (2) 269121 Fax: (39) (2) 21072020 Telex: 340321 I Customer Service (Prices, new product info, product literature): (39) (2) 26901359 Bulletin Board: (39) (2) 21072051 Technical Support: (39) (2) 26901351

Area	Telephone Numbers
Japan	Microsoft Company Ltd. Phone: (81) (3) 5454-8000 Fax: (81) (3) 5454-7972 Channel Marketing (Pre-sales Product Support) Information Center Phone: (81) (3) 5454-2300 Fax: (81) (3) 5454-7951 Customer Service Phone (Version upgrade/Registration) Phone: (81) (3) 5454-2305 Fax: (81) (3) 5454-7952 Languages: Visual C ++: (81) (3) 5454-2364 Fortran: (81) (3) 5454-2353 Masm: (81) (3) 5454-2360 C: (81) (3) 5454-2361 Quick C: (81) (3) 5454-2362
Korea	Microsoft CH Phone: (82) (2) 531-4500 Fax: (82) (2) 531-1724 Technical Support: (82) (2) 531-4800 Technical Support Fax: (82) (2) 563-5194 Technical Support Bulletin Board Service: (82) (2) 538-3256
Liechtenstein	See Switzerland (German speaking)
Luxembourg	Microsoft NV Phone: (32) 2-7303911 Customer Service: (32) 2-7303922 CompuServe: (32) 2-2150530 (GO MSBEN) Bulletin Board: (32) 2-7350045 (1200/2400/9600 bd, 8N1, ANSI) Technical Support: (32) 2-5133274 (Dutch speaking) (32) 2-5023432 (English speaking) (32) 2-5132268 (French speaking)
México	Microsoft México, S.A. de C.V. Phone: (52) (5) 325-0910 Fax: (52) (5) 280-7940 Customer Service: (52) (5) 325-0911 Bulletin Board Service: (52) (5) 590-5988 (1200/2400 baud, 8 bits, No parity, 1 stop bit, ANSI terminal emulation) Technical Support: Desktop & OS: (52) (5) 325-0912 Developers & Advanced Systems: (52) (5) 237-4800

Area	Telephone Numbers
Netherlands	Microsoft BV Phone: 02503-89189 Customer Service: 02503-77700 CompuServe: 020-6880085 (GO MSBEN) Bulletin Board: 02503-34221 (1200/2400/9600 bd, 8N1, ANSI) Technical Support: 02503-77877 (Dutch speaking) 02503-77853 (English speaking)
New Zealand	Microsoft New Zealand Ltd Phone: 64 (9) 358-3724 Fax: 64 (9) 358-3726 Technology Link Centre (Technical Support) Phone: 64 (9) 357-5575 Fax: 64 (9) 358-0092
Northern Ireland	See United Kingdom
Norway	Microsoft Norway AS Phone: (47) (22) 18 302 25 00 Microsoft Sales Support: (47) 22 02 25 80 Microsoft BBS: (47) 22 18 22 09 (Document 404040 in FaxSvar contains detailed instructions) Microsoft FaxSvar: (47) 22 02 25 70 Technical Support: (47) (22) 02 25 50
Papua New Guinea	See Australia
Paraguay	See Argentina
Portugal	MSFT, Lda. Phone: (351) 1 4412205 Fax: (351) 1 4412101
Republic of China	Microsoft Taiwan Corp. Phone: (886) (2) 504-3122 Fax: (886) (2) 504-3121 Technical Support: (886) (2) 508-9501
Republic of Ireland	See United Kingdom
Scotland	See United Kingdom
South Africa	Microsoft South Africa Phone: (27) 11 444 0520 Fax: (27) 11 444 0536
Spain	Microsoft Iberica SRL Phone: (34) (1) 804-0000 Fax: (34) (1) 803-8310 Technical Support: (34) (1) 803-9960

Area	Telephone Numbers
Sweden	Microsoft AB Phone: (46) (8) 752 56 00 Sales Support: (46) (8) 752 56 30 Microsoft FaxSvar: (46) (0)8 752 29 00 Microsoft BBS: (46) (8) 750 47 42 (Document 202020 in FaxSvar contains detailed instructions) Information on Technical Support: (46) (8) 752 09 29
Switzerland	Microsoft AG Phone: 01-839 61 11 Fax: 01-831 08 69 Prices, updates, etc.: 01/839 61 11 CompuServe: GO MSEURO(Microsoft Central Europe) Documentation: Phone: 155 59 00 Fax: 064-224294, Microsoft Info-Service, Postfach, 8099 Zürich Technical support: C/C++, FORTRAN, Macro Assembler PDS: 01/342-4036 Technical support (French speaking): 022-738 96 88
Turkey	Microsoft Turkey Phone: (90) 212 2585998 Fax: (90) 212 2585954 Works for MS-DOS, Works for Windo, FORTRAN, Macro Assembler: 01-342-4036 BASIC, Visual Basic: 01-342-4086
United Kingdom	Microsoft Limited Phone: (44) (734) 270000 Fax: (44) (734) 270002 Upgrades & Registration: (44) (81) 614 8000 Technical Support: Main Line (All Products): (44) (734) 271000 Fax Information Service: (44) (734) 270080 Bulletin Board Service: (44) (734) 270065 (2400 Baud) (44) (734) 270060 (9600 Baud) MSDOS 90 day F.O.C. Support: (44) (734) 271900 MSDOS Charged Support: (44) (891) 315500
Uruguay	See Argentina

Area	Telephone Numbers
Venezuela	Corporation MS 90 de Venezuela S.A. Other information: 58.2.910008 58.2.914739 58.2.913342 Fax: 58.2.923835 Technical Support: 58.2.910046 58.2.910510
Wales	See United Kingdom

Microsoft TechNet, Technical Information Network

Microsoft TechNet is an annual information service created for those who support or educate end users, administer networks or databases, create automated solutions and recommend or evaluate information technology solutions. Microsoft TechNet gives you the answers you need, when you need them by providing timely, comprehensive and in-depth information on Microsoft products and on supporting and administering Microsoft-based solutions.

TechNet members receive twelve monthly TechNet CDs containing the complete Microsoft Knowledge Base, Resource Kits, products facts and features, educational materials, conference session notes, and other technical information. Members also receive four quarterly updates of the Microsoft Drivers and Patches CD containing drivers and patches for most Microsoft applications. Other membership benefits include a dedicated TechNet CompuServe forum (GO TECHNET), and the on-line Microsoft Services Directory.

Availability and the services of Microsoft TechNet may vary by country. To join Microsoft TechNet outside the U.S. and Canada, call the appropriate number listed below:

Country	Telephone Number
Australia	61 2 870 2100
Argentina	54-1-814-5105/54-1-814-4807/54-1-814-4808
Brazil	(55) 011-542-4781
Caribbean Region	(1) (800) 344-2121 or (1) (402) 691-0173
Canada	(1) (800) 344-2121
Chile	65-2-2048257
Colombia	(57) (1) 618-2245
Europe	+31 10 258 88 64
France	05 90 59 04 (toll-free)
Germany	0130 81 02 11 (toll-free)

Country	Telephone Number
Hong Kong	(852) 756-5560
India	91-11-646 0694
Israel	(972) (3) 575-7034
Japan	03-5600-5036
Mexico	(525) 325-09-11
Netherlands, The	06 022 24 80 (toll-free)
New Zealand	64 9 308 9318
Russia	7-095-244-3474
Singapore	(65) 220-7380
South Africa	27 11 445 0000
South Korea	82 2 531 4500
Turkey	90-212-258 59 98
United Arab Emirates	971 4 513888
United States	(1) (800) 344-2121
Venezuela	58-2-910008
All Other Countries	(1) (402) 691-0173

Index

* 5 7 1 5 4 *

Recyclable

Microsoft®